1

January 1992

MTR11227
Volume 1

Richard A. Marcotte
Maurine J. Neiberg
Richard L. Piazza
Lester J. Holtzblatt

Diagnostic Reasoning within Sequential Circuits

CONTRACT SPONSOR ESD
CONTRACT NO. F19628-89-C-0001
PROJECT NO. 702A
DEPT. G111

MITRE

The MITRE Corporation
Bedford, Massachusetts

Department Approval: [illegible]

MITRE Project Approval: [illegible]

Richard A. Marcotte

ABSTRACT

A model-based diagnostic reasoning system makes use of a design model of the structure and behavior of a circuit to diagnose faults within the circuit. In this volume of the FY91 final report for MOIE Project 7020, we describe the Generic Model-based Diagnostic System (GMODS), which can use VHSIC Hardware Description Language (VHDL) models to diagnose single and multiple faults within digital sequential circuits. This volume describes the overall GMODS approach, contrasts the capabilities and processing performance of two GMODS diagnostic algorithms (a minimal envisionment algorithm and a conflict set algorithm), and concludes that the conflict set algorithm is most appropriate for diagnosing sequential circuits.

ACKNOWLEDGMENTS

Barbara Brown, David Krieger, Judy Powell, Michael Sayko, and Kimberly Warren have made substantial contributions to the development of GMODS. Ed Green, Robert Labonté, Joel Schoen, and John Woodward have provided guidance and advice over the course of the year. Marc Vilian has assisted in developing our formalism for temporal methods. Robert Colson and Richard Sveyda of Warner Robins Air Logistics Center and Timothy Szczerbinski of MITRE have helped to identify suitable test cases to support follow-on GMODS development. Connie Roy has assisted in the preparation of this report.

The work described in this report was performed as a MITRE Mission Oriented Investigation and Experimentation (MOIE) effort with oversight provided by Rome Laboratory. During FY91, William Russell and Lt. Tonette Pettinato of Rome Laboratory were the Air Force project officers for this effort and provided helpful guidance and advice.

TABLE OF CONTENTS

LIST OF FIGURES

SECTION 1

INTRODUCTION

The task of achieving an adequate diagnostic capability within complex computer and electronic systems has long been recognized as problematic. Typically, system diagn 'stics are developed to isolate a set of faults that has been listed within a fault isolation table. When the system design changes, the costs associated with maintaining the fault isolation tables, redesigning the Built-in Test (BIT) circuitry, and maintaining test program sets can be prohibitive. In addition, since these diagnostic elements are designed to isolate only the faults listed in the dictionary, they are usually unable to isolate multiple faults and other unanticipated faults. Evaluating the effectiveness of a diagnostics design can also be difficult; there is a lack of automated analysis techniques available to system designers for evaluating fault tolerance or for assessing the extent to which faults can be detected and uniquely diagnosed.

1.1 THE ROLE OF HARDWARE MODELING

These problems can be partly mitigated through the use of the VHSIC Hardware Description Language (VHDL) and emerging analog and system-level description languages. In particular, VHDL has been mandated as a DOD documentation requirement for military Application-specific Integrated Circuits (ASICs) and DOD Standard Parts. These design description languages can provide a standard medium for sharing candidate designs with the test organization, allowing test developers to influence testability aspects of candidate designs at an early stage in the design process.

Nonetheless, additional efforts are required to fully address the problems cited above. Although VHDL and other standard descriptions are becoming available, there is a lack of automated analysis techniques that can use these descriptions to evaluate fault tolerance and fault detection and isolation coverage. Additionally, conventional fault simulation and test development approaches still depend on enumerating stuck-at faults. Two "model-based" design and diagnostics technologies which could help to address these remaining difficulties include model-based diagnostic reasoning and behavioral-level fault simulation.

Model-based diagnostic reasoning techniques can improve fault isolation coverage. Rather than isolating only faults that were enumerated in advance, a model-based diag[illegible] [illegible]stem uses a design model to produce diagnostic explanations for the observe[illegible] [illegible]eh[illegible]
Since any deviation from working behavior is considered a fault, [illegible]
diagnose unanticipated faults and multiple faults. They are also [illegible]

rather than having to create and maintain test program sets or rules, design models can be exploited directly to support subsequent troubleshooting activities.

Since they use design models, model-based reasoning techniques can also be used during the circuit design process to evaluate fault tolerance and fault isolation coverage. By identifying the combinations of component faults that can contribute to an undesirable system outcome, a model-based diagnostic system can perform an automated fault tree analysis directly from the design model. By determining the extent to which fault isolation tests lead to unique diagnoses, it can also evaluate the fault isolation coverage to be provided by the diagnostics elements of the system design.

Behavioral-level fault simulation techniques can be used as a design tool to evaluate system performance in the presence of faults and to evaluate fault detection coverage. Unlike conventional fault simulation techniques which are generally limited to considering stuck-at faults within gate-level hardware models, behavioral-level fault simulation techniques exploit the descriptive capabilities of VHDL to incorporate additional fault classes (bridging faults, etc.) into a fault simulation, and to allow circuit designers and test developers to define new fault classes based on operational experience. By supporting additional fault classes in this manner, testability approaches can be validated for a broader range of possible faults.

1.2 PROJECT OVERVIEW

The goal of MOIE Project 7020, Model-based Diagnostic Reasoning, is to develop both model-based diagnostic reasoning and behavioral-level fault simulation techniques that can be used in conjunction with VHDL. Our primary emphasis has been to develop a prototype of the Generic Model-based Diagnostic System (GMODS) that is able to diagnose faults by reasoning about discrepancies between the behavior of a faulted circuit and the behavior of a VHDL design model of the circuit. Since any deviation from designed behavior is considered a fault, GMODS has the potential to diagnose unanticipated faults as well as performance degradations indicative of an impending fault. Since it reasons from VHDL design models, GMODS can also be used as a design tool to evaluate fault isolation coverage.

In addition to GMODS, the project has also developed a behavioral-level fault simulation technique called signal faulting. This technique is based on envisioning the effects of likely hardware failure mechanisms on signals within a VHDL model. Each signal is divided into a origi[illegible] signal and a [illegible]al, possibly faulted signal, and a selectable fault transformation [illegible] possibly faulted signals to model the effects of hardware failure [illegible]) on the behavior of the signal. New classes of faults (such as [illegible]ys) can be easily introduced by defining new fault transformations. The signal faulting approach is documented within a separate volume of the FY91 final report for Project 7020.

1.3 OBJECTIVES FOR GMODS

Our primary objective for GMODS has been to develop a general-purpose model-based diagnostic system that can diagnose single and multiple faults at the board level within digital circuits. In this paper, we will define a *diagnosis* to be an assertion that a set of zero or more circuit components are in a faulted state. For a circuit that has **n** faultable components, where each component is either working or faulted, at least 2^n possible diagnoses can potentially be asserted. Given a set of observations from the faulted circuit, the overall task for GMODS is to search this space of possible diagnoses in an efficient manner to determine which diagnoses are consistent with the observations, and to determine which diagnoses to report to the user.

To be able to diagnose faults within realistic digital circuit boards, the GMODS approach must satisfy the following design goals:

- Fault coverage. GMODS must avoid making restrictive assumptions about the classes or combinations of faults that can occur within the system. For example, it should not be limited to diagnosing only single faults or diagnosing only fault modes envisioned in advance.

- Plausibility. GMODS must avoid computing and reporting relatively improbable diagnoses when more likely ones are consistent with the observations. Similarly, GMODS should avoid reporting suspect components whose likely fault modes are inconsistent with the observations.

- Efficiency. GMODS must be able to compute the set of plausible diagnoses with a processing time that has a linear or polynomial relationship to the number and complexity of circuit components.

- Generality. GMODS must be able to handle realistic component models and circuit topologies. In particular, it must also be able to reason about the behavior of sequential circuit elements such as memory elements and feedback mechanisms.

1.4 CONTENTS OF THIS PAPER

This volume of the FY91 final report for Project 7020 will describe the GMODS approach, with a focus on describing the GMODS reasoning mechanisms and on evaluating two diagnostic reasoning algorithms that have been incorporated into GMODS. Section 2 will provide an overview of previous efforts in model-based diagnostic reasoning, with an emphasis on the previous diagnostic algorithms which provided a basis for the GMODS effort. Section 3 provides an overview of the GMODS reasoning mechanisms and the diagnostic algorithms incorporated into GMODS. Section 4 then provides more detail on the GMODS representation language; section 5 discusses the GMODS temporal reasoning mechanisms to support diagnostic reasoning over time; and section 6 describes the GMODS constraint propagator. Sections 7 and 8 provide detailed descriptions of the GMODS diagnostic algorithms, and section 9 provides a comparison of the two diagnostic algorithms. Finally, section 10 provides conclusions from the effort.

SECTION 2

PREVIOUS WORK

Model-based diagnostic reasoning has been the subject of extensive research over the past decade [2, 5, 6, 7, 8, 9, 10, 11, 13, 15, 16, 17, 18]. Much of this work has been based on the use of constraint propagation techniques [19] for building circuit and system models in a declarative style intended to support qualitative simulation, hardware verification, design synthesis, and diagnosis.

2.1 MODELING BEHAVIOR USING CONSTRAINTS

Within a constra: formalism, a *constraint* is used to represent a local relationship between a set of objects (such as variables or nodes) that is contained within a larger space of objects. For example, the expression 9 * C = 5 * (F - 32), which relates temperature expressed in degrees Farenheit to temperature expressed in degrees Centigrade, could be expressed in terms of the following constraints:

* (C, 9, Templ)
* (Temp2, 5, Templ)
+ (Temp2, 32, F)

The constraints and the variables that they relate can be used to form a constraint network. The two primary types of elements in a constraint network are the variables and the constraints which define relationships among the variables. Constraints serve two purposes: they express the relationships between variables, and they can be used to generate values for each of the variables. In the above, if the value of C is known to be 10, the value of Templ can be computed, because if C, Templ and 9 are to satisfy the * relation, then the only value Templ can be assigned is 90.

A constraint propagation algorithm endeavors to assign values to all of the objects within the space, such that all of the relationships expressed by the constraints are satisfied. Given a constraint that expresses a relationship between n variables within a constraint network, the constraint propagator may be asked to infer a value for one of the variables, where the n-1 variables are the antecedents of the inference and the nth variable is the consequent. Procedural information can be associated with each type of constraint to enable the constraint propagator to generate an assignment for the nth variable in a relation, based on the values of the other n-1 variables. For example, if the value of F is known in the constraint network above, we can use one of the procedures associated with the constraint "+" to compute the value of Temp2.

Constraint formalisms can be used to model the behavior of digital electronic hardware in a declarative format that is particularly well suited for diagnosis. Unlike a more procedural representation of behavior, constraints can be used to infer values in both the intended direction of information flow (from inputs to outputs) and in the opposite direction (from outputs to inputs). This is often useful in diagnosis; when given an observed value for a circuit output, it is often desirable to infer the possible input values that could explain the observed value. When using a constraint formalism to model digital circuits that have an intended direction of information flow, *forward propagation* refers to the process of constraint propagation in the intended direction. Likewise, *back propagation* refers to the process of constraint propagation in the opposite direction, from circuit outputs toward inputs.

2.2 USING CONSTRAINT MODELS TO SUPPORT DIAGNOSIS

An early approach for using constraint models to support diagnostic reasoning is described by Davis [2]. His approach was to model a circuit both structurally, as a network of replaceable components, and behaviorally, as a set of constraints associated with each component. Given a set of inputs to the circuit, the structural model, and the constraints on each component, simulated values for each circuit output are computed. If these values differ from the real outputs, the circuit is believed to be faulty and a diagnostic algorithm is invoked. To isolate the fault, Davis proposed a generate and test algorithm called *constraint suspension*: identify constraints within the model that, if suspended, cause all of the discrepancies between observed and expected behavior to be removed. Davis used this technique to isolate single faults within simple combinational circuits.

Another early model-based reasoning system, the Liquid Oxygen (LOX) Expert System (LES), was developed to diagnose faults within the space shuttle LOX propellant loading system [17]. LES is able to diagnose faults using a model of the steady-state behavior of over 1,000 analog and digital electromechanical components within the LOX loading system. Although it was run successfully during several space shuttle launches, like Davis' approach it was limited to diagnosing single faults, and the developers of LES were required to approximate the LOX loading hardware as a combinational circuit.

2.3 APPROACHES FOR DIAGNOSING MULTIPLE FAULTS

A limitation of these early systems was their inability to diagnose multiple faults in an efficient fashion. For example, the logical extension to LES to handle multiple faults would be to generate and test all single and multiple fault suspects, which is computationally intractable. In 1987, de Kleer and Williams published a description of their General Diagnostic Engine (GDE), which incorporated a diagnostic algorithm that can diagnose both single and multiple faults without generating and testing all possible diagnostic hypotheses [5].

The de Kleer/Williams approach relies on the principle of *parsimony*, which is to consider only the simplest set of diagnostic hypotheses that satisfy all of the circuit observations. A parsimonious set of diagnostic hypotheses is one that contains only hypotheses which cannot be subsumed by any other hypothesis in the set. Any diagnostic hypothesis that is a superset of a parsimonious hypothesis is also valid, and therefore, the parsimonious set of diagnostic hypothesis provides a compact representation of the complete set of possible diagnostic conclusions.

The de Kleer/Williams algorithm computes the most parsimonious suspects directly using the following steps:

- It propagates each observed input and output value through the model, based on assumptions that each component is working.
- Each value generated during constraint propagation is labeled with a parsimonious set of environments. An environment is a set of assumptions under which a value can be generated.
- It finds environments (called conflict sets) that lead to contradictions between the behavior of the circuit and that of the model. A contradiction occurs when two different values are inferred for the same node. A conflict set is generated by computing the cross product[1] of the labels of the contradicting values, and then eliminating non-parsimonious environments.
- It computes parsimonious diagnoses by taking successive cross products of the conflict sets.

To illustrate, consider the simple multiplier/adder circuit in figure 1, where outputs Y and Z are both faulted. Based on the assumptions that the multiplier and adders are working, we can forward propagate the input values A = 2 and B = 3 to infer the values X = 6, Y = 10, and Z = 10. Each inferred value is labeled with the supporting assumptions:

X = 6 (M1-working)
Y = 10 (M1-working, A1-working)
Z = 10 (M1-working, A2-working)

We can observe discrepancies between inferred and observed values at both Y and Z. Consequently we can identify two conflict sets, (M1-working, A1-working) and (M1-working, A2-working), which each represent a set of assumptions that leads to a contradiction. The conflict sets can be alternatively expressed as (M1-faulted or A1-faulted) and (M1-faulted or A2-faulted).

[1] For example, the cross product of the sets {{a}, {b, c}} and {{d, e}, {f}} is the set {{a, d, e}, {a, f}, {b, c, d, e}, {b, c, f}}.

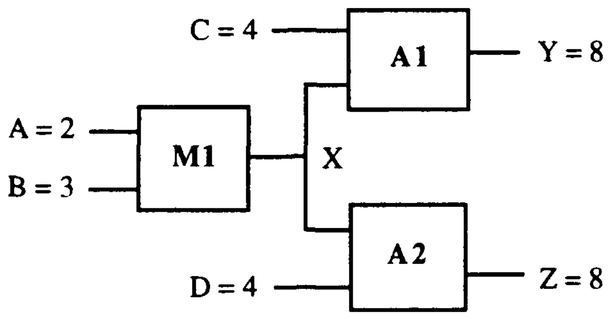

Figure 1. A Simple Diagnostic Example

We can now compute the parsimonious diagnoses by converting the conflict sets into disjunctive normal form using a cross product operation. This yields:

(M1-faulted) or
(A1-faulted, A2-faulted)

which tells us that M1 failing alone, or A1 and A2 failing together, can explain the observations. We could also identify (M1-faulted, A1-faulted, A2-faulted) or (M1-faulted, A1-faulted) or (M1-faulted, A2-faulted) as possible diagnoses. However, since each of these diagnoses can be subsumed by one or more of the diagnoses reported above, they are less parsimonious.

In a second scenario where Y = 8 and Z = 6, we can generate a third conflict set at node X using back propagation. First, we can back propagate from Y = 8 to compute X = 4 under the assumption A1-working. We can then back propagate from Z = 6 to compute X = 2 under the assumption A2-working. Since X cannot be equal to both 4 and 2 simultaneously, we identify a third conflict set (A1-working, A2-working). Thus:

(M1-faulted or A1-faulted) and
(M1-faulted or A2-faulted) and
(A1-faulted or A2-faulted),

which leads to the parsimonious diagnosis:

(M1-faulted, A1-faulted) or
(M1-faulted, A2-faulted) or
(A1-faulted, A2-faulted).

By finding conflict sets and converting them directly to parsimonious suspect sets, the de Kleer/Williams algorithm is able to avoid having to generate and test all possible diagnostic conclusions. Nonetheless, cross product operations are exponential, and the task of converting the conflict sets into diagnoses in particular can require extensive processing time, especially for highly interconnected circuits with limited observability.

In 1987, Hector Geffner and Judea Pearl proposed a diagnostic algorithm that employed an alternative technique for managing the size of the search space [8]. Rather than applying the principle of parsimony, they proposed using minimal cardinality to guide the diagnostic process (i.e., they proposed to find all of the diagnoses that depend on the minimal *number* of faults).

The Geffner/Pearl algorithm uses a circuit model in which each component has two modes of operation, a working mode and a default fault mode. The working mode includes a set of constraints that define the working behavior of the component. The default fault mode simply states that the outputs of a component are undefined for any combination of inputs. (We will use the symbol "?" to refer to the set of possible values predicted using the default fault mode. Since the ? can represent any possible value, it is frequently treated by the Geffner/Pearl algorithm as being consistent with actual values.) In addition, the Geffner/Pearl algorithm can be easily extended to use explicit fault modes that define fault behaviors for various components.

The effects of each observed input and output value are then propagated through the model, using both the working and fault modes of operation for each component. Each value generated by the constraint propagator is tagged with sets of justifications and weights. A *justification* contains a set of antecedents and the mode of operation of the constraint used to generate the value. A *weight* represents the minimal number of fault mode assumptions on which the value depends. Since values at internal nodes can be derived from more than one direction (as a result of back propagation), a different weight is computed for each direction, and the justifications are partitioned by direction. By tagging each value in this manner, a dependency graph is constructed of the justifications that support each of the predicted values.

After the constraint propagator has predicted the effects of all of the input and output values on the model, diagnoses that explain the observed output values can then be generated. Generating a diagnosis involves two steps:

- identifying predicted values consistent with the observed values, and

- tracing through the justifications for the consistent values that have the lowest weight, to identify the component modes of operation which explain the observations.

For example, consider the diagnostic scenario shown in the circuit of figure 1. To compute a diagnosis using the Geffner/Pearl approach, we first define two modes of operation for each component, the working mode and the default fault mode. The constraint propagator first forward propagates values through the model to generate predicted values for each intermediate node and output node. By propagating from the inputs, two values are derived for X:

Value	Direction	Weight	Justifications
X = 6	M1-out	0	[Justification: A = 2, B = 3, M1-working]
X = ?	M1-out	1	[Justification: A = 2, B = 3, M1-faulted]

Similarly, two values are derived for Y:

Value	Direction	Weight	Justifications
Y = 10	A1-out	0	[Justification: C = 4, X = 6, A1-working]
Y = ?	A1-out	1	[Justification: C = 4, X = ?, A1-working] [Justification: C = 4, X = 6, A1-faulted]

Notice that a third possible justification for Y = ?, [Justification: C = 4, X = ?, A1-faulted], has not been included because it is not minimal. In addition, a set of values, weights, and justifications are computed for Z.

The second step is to back propagate values within the model to take account of the observed values at the outputs. If the observed value of Z is 8, then a value of X = 4 would be derived and additional justifications would be generated for X = ?. At this stage, the justifications and weights for values at X include:

Value	Direction	Weight	Justifications
X = 6	M1-out	0	[Justification: A = 2, B = 3, M1-working]
X = ?	M1-out	1	[Justification: A = 2, B = 3, M1-faulted]
X = 4	A2-in1	0	[Justification: D = 4, Z = 8, A2-working]
X = ?	A2-in1	1	[Justification: D = 4, Z = 8, A2-faulted]

Finally, the value X = 4 is propagated to Y (generating a value Y = 8), and the weights for the values at Y are updated. Afterward, the minimal justifications and weights for values at Y are:

Value	Direction	Weight	Justifications
Y = 10	A1-out	0	[Justification: C = 4, X = 6, A1-working]
Y = 8	A1-out	1	[Justification: C = 4, X = 4, A1-working]
Y = ?	A1-out	2	[Justification: C = 4, X = ?, A1-working] [Justification: C = 4, X = 6, A1-faulted]

To compute the weights for the values at Y, it is necessary to compute the weights contributed by each antecedent. This involves visiting each antecedent node, determining the minimal weights of consistent values inferred from each direction to the node, and then adding these weights together.[2] For example, Y = 8 has one justification with three antecedents: C = 4, X = 4, and A1-working. The antecedents C = 4 and A1-working both contribute a weight of zero. To determine the weight contributed by the antecedent X = 4, node X must be visited. At node X, two values are considered to be consistent with the antecedent X = 4, the value X = 4 derived from the direction A2-in1, and the value X = ? derived from the direction M1-out. (The value X = ? derived from A2-in1 is also consistent, but it is not considered because it has a higher weight than the value X = 4 derived from the same direction.) The overall weight for the antecedent X = 4 is computed by adding the weights for the minimal consistent values from each direction; X = 4 from A2-in1 has a weight of zero, X = ? from M1-out has a weight of one, and the antecedent X = 4 contributes the sum, a weight of one, to the consequent value Y = 8.

After constraint propagation is completed, the final step is to trace through the dependency graph to identify the sets of assumptions that underlie the values consistent with the observations. By examining node Y, we see that two values are consistent with the observations: Y = 8 and Y = ?. However, in this case, we can ignore Y = ?, since it has a higher weight. By tracing the justifications for Y = 8, we obtain the set of supporting assumptions (A1-working, A2-working, M1-faulted), which is the minimal diagnosis for the circuit. Notice that as a result of back propagation, the justification network for Y = 8 already includes support from Z = 8, and so the diagnosis for Y = 8 represents the overall diagnosis for the circuit. In another scenario, if the observed output values are Y = 0 and Z = 8, the values, justifications, and weights derived for Y would be identical. By examining node Y, we see that the only value consistent with the observations is Y = ?, and by tracing the justifications for Y = ?, we arrive at the diagnosis (A1-faulted, A2-faulted, M1-working).

Unlike generating diagnoses by taking cross products of conflict sets, Geffner and Pearl claim that the process of tracing the network of justifications to produce a diagnosis is computationally

[2] As noted in [8], this algorithm for computing weights from the antecedents is somewhat more complex when reconvergent fanout is present within the circuit model.

linear (in the absence of reconvergent fanout). Their algorithm therefore represents an important advance in model-based diagnostic reasoning. However, it still does not address all of the goals listed earlier; in particular, it does not attempt to reason about behavior over time within sequential circuits.

SECTION 3

AN OVERVIEW OF GMODS

None of the previous approaches are able to satisfy all of our goals for a model-based diagnostic system. In particular, none are able to diagnose sequential circuits, ones that contain internal memory elements or feedback mechanisms, and none make use of standard design descriptions. To demonstrate that model-based diagnostic reasoning techniques could be applied to standard design descriptions of digital sequential circuits, we have developed a GMODS prototype that is able to convert VHDL design descriptions of digital sequential circuits into GMODS constraint models, and can then diagnose faults by reasoning about the behavior of the circuit models over successive clock intervals.

3.1 REASONING MECHANISMS

As shown in figure 2, GMODS consists of the following components: a circuit model defined using the GMODS representation language; a constraint propagator that can propagate observed values within the model based on various diagnostic assumptions; a reason maintenance system that keeps track of the assumptions and time intervals which support each value inferred by the constraint propagator; and a diagnostic problem solver that uses the constraint propagator and reason maintenance system to produce single and multiple fault diagnoses. GMODS also includes a translator that is able to convert VHDL descriptions into GMODS models in an automated fashion.

A GMODS circuit model contains both structural and behavioral information. A circuit is represented structurally by a network of components connected by idealized connectors called nodes. A set of behavioral models, called modes of operation, is associated with each component. The modes of operation can include a working mode, some number of explicit fault modes, and a default fault mode in which the outputs are unpredictable for any combination of inputs. For each mode of operation, the component's behavior is represented by a set of constraints, where each constraint imposes a relationship among state variables that represent the component's inputs, outputs, and internal state elements. The relationship defined by a constraint has two key aspects: *predicates* define a method of computing values for each state variable referenced by the constraint, and *temporal methods* define the temporal extents of the values computed for each state variable.

The constraint propagator functions much like a simulator. Starting with values specified at the primary input nodes, the constraint propagator uses the constraint predicates to make local inferences at each component, computing the component outputs based on the data at the inputs. These inferences generate data at other state variables, making other inferences possible. In this way, data propagates through the model from the primary inputs to the primary outputs. Since each local inference represents the behavior of a component based on values at its input ports, these inferences taken as a whole represent a simulation of the circuit. Additionally, given observed values at the primary outputs, the constraint propagator can back propagate these values to internal nodes as necessary to support the diagnostic reasoning process.

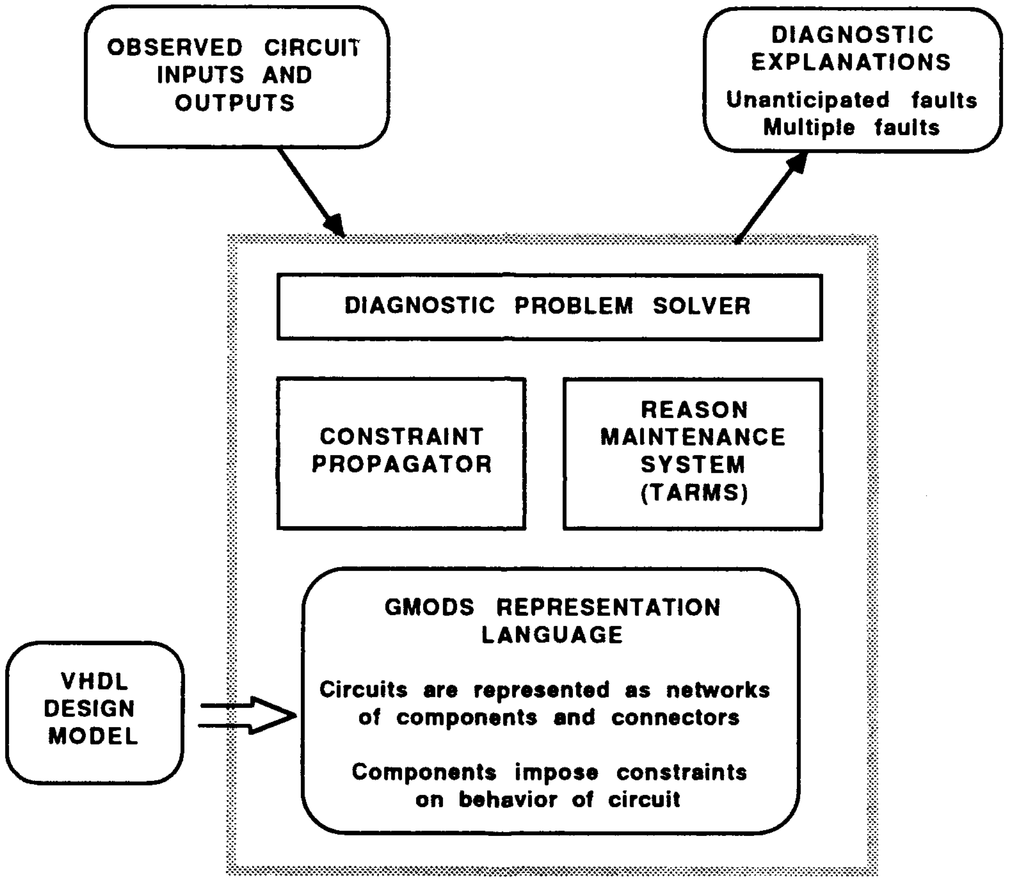
OBSERVED CIRCUIT INPUTS AND OUTPUTS
DIAGNOSTIC EXPLANATIONS
Unanticipated faults
Multiple faults
DIAGNOSTIC PROBLEM SOLVER
CONSTRAINT PROPAGATOR
REASON MAINTENANCE SYSTEM (TARMS)
GMODS REPRESENTATION LANGUAGE
Circuits are represented as networks of components and connectors
Components impose constraints on behavior of circuit
VHDL DESIGN MODEL

Figure 2. An Overview of the GMODS Architecture

As the constraint propagator generates values for state variables within the model, GMODS maintains a record of the sets of assumptions which support each value, the number of fault mode assumptions (the weight), and the time intervals in which the value is valid under the indicated set of assumptions. To perform this bookkeeping, we developed the Temporal Assumption-based Reason Maintenance System (TARMS), which extends previous work in assumption-based truth maintenance [3] to include temporal reasoning. Over time, the constraint propagator may infer several values for a given state variable within the model. For each value, TARMS uses the temporal method to compute the time intervals during which the value is supported, and then computes the sets of supporting assumptions for each of these time intervals. The time intervals and sets of supporting assumptions for each value are maintained within an *assumption history.*

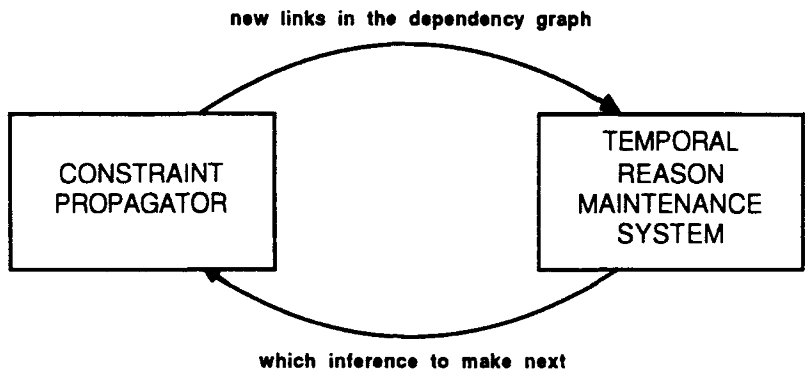
new links in the dependency graph
CONSTRAINT PROPAGATOR
TEMPORAL REASON MAINTENANCE SYSTEM
which inference to make next

Figure 3. The Constraint Propagator and TARMS Work in Tandem

Under the control of the diagnostic problem solver, the constraint propagator works in tandem with TARMS to produce a diagnosis (see figure 3 above). The constraint propagator propagates all of the observed input and output values through the circuit model, generating values for state variables in the model. While this activity takes place, TARMS constructs a dependency graph (where the "nodes" in the graph represent datum values and the "arcs" represent justifications) and maintains the sets of supporting assumptions for each datum. The constraint propagator indicates how the graph should be constructed based on the forward and backward inferences made. Links are made in the dependency graph between datum values at the antecedents to the datum value at the consequent. If a datum can influence the explanation of another datum, then they are connected through some path in the dependency graph.

The constraint propagator is relatively independent of temporal considerations; the temporal methods associated with the constraints are used by TARMS to determine if and when a consequent datum is valid. The constraint propagator, having computed the consequent, need never revisit the inference. This allows the constraint propagator to perform the actual value propagation only once, leaving it to TARMS to perform assumption propagation to update the explanation.

TARMS maintains the consistency of the information at each node in the dependency graph, and suggests which inference the constraint propagator should perform next. When the explanation for an antecedent datum changes, the explanation for the consequent changes using assumption propagation techniques. If the consequent datum's explanation changes, *its* consequents must also be updated. This happens in a recursive manner and is facilitated by following the links in the dependency graph. Assumption propagation continues until either there are no consequent justifications or the explanation of a consequent does not change.

3.2 DIAGNOSTIC ALGORITHMS

The GMODS diagnostic problem solver incorporates two diagnostic algorithms: a conflict set algorithm similar to [5] and a minimal envisionment algorithm similar to [8]. By implementing both algorithms using the same underlying reasoning mechanisms, we have been able to examine the effectiveness and relative performance of these algorithms for reasoning over time within digital sequential circuits.

The minimal envisionment algorithm was defined by combining the Geffner/Pearl algorithm with the assumption tracking and temporal reasoning capabilities of TARMS. From the original Geffner/Pearl algorithm, we have retained the concept of a minimal diagnosis, the technique of pruning non-minimal justifications, and the technique of always propagating the effects of the default fault mode. By implementing the algorithm using TARMS, we have added the capability to label values with sets of assumptions and time intervals, which permits the algorithm to reason over multiple clock intervals, to support incremental diagnostic reasoning, and to handle reconvergent fanout and feedback topologies. In addition, since the sets of supporting assumptions for each value are computed during constraint propagation, all computation needed to derive the diagnosis takes place during propagation; unlike the original Geffner/Pearl algorithm, no further tracing of the dependency graph is required to produce a diagnosis.

The conflict set algorithm is a variation of the original de Kleer/Williams algorithm that has been implemented using TARMS to support reasoning over time within sequential circuits. When running the conflict set algorithm, GMODS propagates only working mode behavior. When two values at a state variable have sets of supports which overlap in time, a contradiction is detected and the contradictory sets of supports are combined to produce a conflict set. The conflict sets are subsequently used to generate the single fault diagnoses using an intersection operation, or to produce a more complete parsimonious diagnosis using the original de Kleer/Williams approach.

Based on the selection of the diagnostic algorithm, the diagnostic problem solver makes adjustments to the operation of the constraint propagator and TARMS. Constraint propagation adjustments are made to control whether the default fault mode is propagated, and to prevent unnecessary inferences that can otherwise be made using the default fault mode. Likewise, TARMS makes use of a different method for combining sets of supports; to support the conflict algorithm, TARMS computes the most parsimonious sets of supports, and to support minimal envisionment, TARMS computes the minimal sets of supports. These and other differences will be detailed in sections 7 to 9.

SECTION 4

MODELING TO SUPPORT DIAGNOSIS

The GMODS representation consists of three primary elements: a structural representation of the circuit as a network of components and idealized connectors called nodes, a representation of the material and information flow over time within the circuit, and a description of the behavior of each component. The GMODS representation of a digital sequential circuit can be defined in an automated fashion from a corresponding VHDL description of the circuit.

4.1 REPRESENTING CIRCUIT STRUCTURE

The overall structure of a circuit is represented by a GMODS network. A network defines the connectivity among a set of network components and the primary network inputs and outputs. Within the network, a component can represent any replaceable circuit elements to which faults can be attributed, including replaceable chips on a circuit board, wires and busses, logic gates, and so forth. A component has a number of ports that represent interfaces through which materials or information can flow, and it may have other attributes that represent internal state attributes, parameters, and local variables.

Within the network, components are interconnected by attaching the appropriate component ports to a common node. As in electronic circuit theory, a node represents an ideal connector. Nodes are also used to represent the primary input and outputs of the network.

4.2 REPRESENTATION INFORMATION FLOW

State variables maintain the values that are inferred for nodes within a network and for internal state attributes associated with various components. State variables also serve as an interface between the model and TARMS; the assumptions, justifications, and time intervals under which a value holds at a state variable are all maintained by TARMS. Each value inferred at a state variable is called a datum.

A state variable may be assigned values of any data type that has been pre-defined within GMODS. The allowable data types currently include integers, ranges, "?", and tri-value vectors. A range defines a set of contiguous integer values. The ? stands for any value within the range of all possible integer values. A tri-value vector is similar to a bit vector, except that each element can take on the value 1, 0, or ?. State variables that maintain tri-value vectors are shown in figure 4.

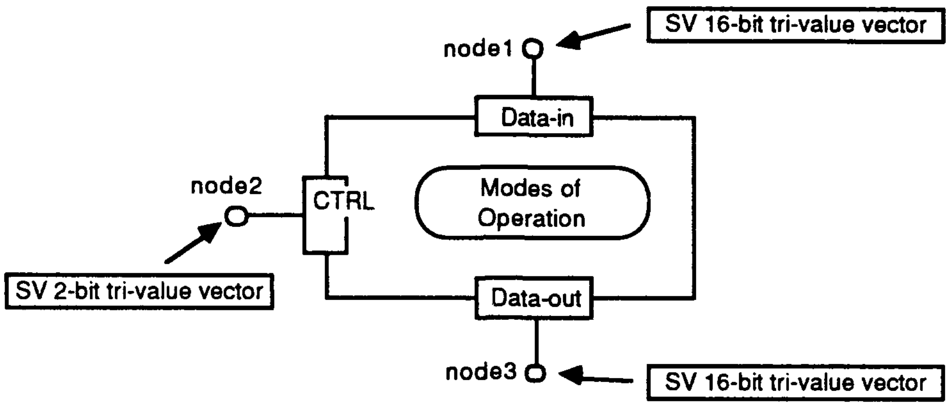
SV 16-bit tri-value vector
node1
Data-in
node2
CTRL
Modes of Operation
SV 2-bit tri-value vector
Data-out
node3
SV 16-bit tri-value vector

Figure 4. A Component Attached to Nodes and State Variables

4.3 REPRESENTING BEHAVIOR

The behavior of each component is described in terms of collection of modes of operation, including a mode of operation called *working* that describes the behavior of a component that is not faulted, some number of explicit fault modes that characterize the behavior of a component when it has become faulted in some known manner, and a default fault model called *faulted* that characterizes the behavior of the component when it has become faulted in an arbitrary, unforeseen manner. The default fault model simply states that the outputs of a component are undefined for any combination of inputs.

For each mode of operation, the behavioral relationships between component inputs, internal state elements, and outputs are represented by a set of *conditional constraints*. In GMODS, a conditional constraint defines a relationship among state variables that is imposed by a particular component. It indicates the component modes of operation for which it applies, and contains a condition clause that specifies values for control signals that must be present before it can be invoked. It also has a consequent clause that specifies the relationship between component input ports, state attributes, and output ports that holds for the mode of operation when the condition clause is satisfied.

Like other constraint formalisms, GMODS constraints can be inverted – not only can a constraint be used to generate values along the causal path from inputs to outputs, but given an observed output and assumptions about some of the inputs, it can also be used to infer the remaining inputs. To support this forward and back propagation, GMODS constraints are described in terms of invertible behavioral primitives called *predicates*. A predicate can be used to test the truthhood of a relationship among state variables, and it can also be used to generate values for one or more of the state variables involved in a relationship.

For example, consider a 16-bit shifter which has two modes of operation, working and the default fault mode. When the shifter is working, it can perform a no-shift, right-shift, or left-shift operation, depending on the value at the function-select port. The constraint definition for the right-shift operation is given by:

```
defconstraint SHIFTER-1 SHIFTER (function-select input output)
   :applicable-modes (working)
   :conditions ((*equal* (integer function-select) '1))
   :consequents ((*right-shift* (16-bit-tri-value-vector output)
                                (16-bit-tri-value-vector input)))
   :causal-flow output
   :delay 0
   :temporal-method intersection
```

The predicate *right-shift* has been used to define the consequent of this constraint. This predicate contains three procedures: one to test whether the *right-shift* relationship holds between the state variables at the input and output nodes, one to generate a value for the output state variable, and one to generate a value for the input state variable. For example, given a 16-bit tri-value vector at the input state variable, a generator attached to the predicate can generate the correct pattern of "1s" and "0s" for the output state variable. Another generator can generate the correct pattern of "1s" and "0s" for the input state variable, except for the rightmost bit, which must be assigned a value of "?".

This constraint also makes reference to the temporal method called intersection which describes the temporal aspects of the constraint. For example, if the function-select and input state variables for the constraint shown above were valid during (t_0, t_4) and (t_1, t_5) respectively, the intersection temporal method would determine that the output would be valid during (t_1, t_4). In section 5, we will provide a more detailed discussion of how various temporal methods are used to model the temporal behaviors of combinational and sequential devices.

4.4 DEFINING A GMODS MODEL

Defining a GMODS model for a circuit requires two principal stages: (a) to construct generic models for the network and for each type of component in the network, and then (b) to use the generic models to instantiate a GMODS network.

In the first stage, the network and the various types of components are described by network descriptors and component descriptors. A component descriptor defines the attributes of the component type and specifies the modes of operation and constraints that will be associated with each instance of the component. For example, the component descriptor for the shifter discussed previously is defined by:

```
defcomponent SHIFTER
        :slots ((function_select :type port)
                (input :type port)
                (output :type port))
        :modes-of-operation (working)
```

As constraints are then defined for each component mode of operation, they are attached to the component model.

A network descriptor describes the types of components in the network, the connectivity among instances of those components, and the primary inputs and outputs. Network connectivity is defined by identifying the ports of component instances to be connected, while the primary inputs and outputs are defined by identifying the ports of component instances that will serve as primary inputs and outputs respectively. For digital circuits, the component descriptors, constraint definitions, and network descriptor can be defined automatically by the VHDL-to-GMODS translator.

In the se[illegible]nd stage, the generic component and network models are used to instantiate a GMODS ci[illegible] model. GMODS first accesses the network descriptor to obtain a listing of the componen[illegible] be instantiated, and then performs the following tasks for each component:

- The component descriptor that defines the specified component type is used to create a component instance.

- Slot specifications accessed from the component descriptor are used to create the ports, parameters, state attributes, and local variables to be associated with the component instance.

After the components in the system have been instantiated, they are connected into a network using the connectivity specification from the network descriptor. A node is created for each connection point and the specified component ports are connected to the node. For each component, the modes of operation and constraints are accessed from the component descriptor, instantiated, and attached to the component. References to each constraint are placed within all of the referenced slots, and state variables are defined to maintain the data that will be generated by the GMODS constraint propagator. Once this activity has been completed, the GMODS model is fully instantiated and ready to support diagnosis.

4.5 TRANSLATING VHDL MODELS TO GMODS

The GMODS network, component, and constraint descriptors can all be defined automatically from corresponding VHDL descriptions. The following VHDL inputs are required to create a GMODS model that can be fully instantiated:

- a top-level VHDL design entity and structural architecture that represents the overall circuit. The interface description for this entity defines the primary circuit inputs and outputs, and components in this structural architecture are assumed to correspond to replaceable components in the circuit.
- a VHDL design entity and dataflow architecture for each type of component referenced in the structural model.
- equivalent VHDL and GMODS libraries of signal types and behavioral primitives.

The GMODS network model is created from the top-level VHDL design entity and structural architecture. The GMODS component models are also created from the corresponding VHDL models using a straightforward mapping. First, the interface description and port declarations of the VHDL design entity are mapped directly into a GMODS component definition form, as shown below:

```
entity SHIFTER is
     port (function_select: in integer;
            input: in bit_vector (15 downto 0);
            output: inout bit_vector (15 downto 0);
  end;

defcomponent SHIFTER
     :slots ((function_select :type port)
            (input :type port)
            (output :type port))
     :modes-of-operation (working)
```

The dataflow architecture is then used to construct sets of GMODS constraints. For combinational components, this translation is also straightforward – each condition of a signal assignment statement leads to another GMODS constraint. However, the behavioral primitives used to define the signal assignment statements (which are usually implemented as VHDL functions) must correspond to GMODS predicates; GMODS does not attempt to construct its predicates automatically from VHDL. For example, consider the following VHDL dataflow description for a 16-bit shifter:

```
architecture dataflow of SHIFTER is
begin
output <= input      when function_select = 00 else
          r_shift (input)      when function_select = 01 else
          l_shift (input);
end;
```

This description gives rise to three GMODS constraints which define the identity operation, the right shift operation, and the left shift operation respectively. The identity constraint is defined by:

```
defconstraint SHIFTER-0 SHIFTER (function-select input output)
    :applicable-modes  (working)
    :conditions  ((*equal*  (2-bit-tri-value-vector  function-select)  '00))
    :consequents  ((*equal*  (16-bit-tri-value-vector  output)
                             (16-bit-tri-value-vector  input)))
    :causal-flow  output
    :delay 0
    :temporal-method  intersection
```

where the both the equality used to define the condition (function_select = 00) and the equality imposed by the signal assignment (output <= input) have been translated to the GMODS *equal* predicate. Similarly, to define the constraints which model the right-shift and left-shift operations, the r_shift and l_shift functions must correspond to the GMODS *right-shift* and *left-shift* predicates. The VHDL-to-GMODS translation process is described more fully in [15].

SECTION 5

TEMPORAL REASON MAINTENANCE

Within a GMODS circuit model, a set of data is associated with each state variable. Each datum in the set represents a value that was inferred by the constraint propagator using one of the constraints attached to the state variable. These values will be inferred based on different sets of diagnostic assumptions about the mode of operation of components in the model and different combinations of values at the antecedents to the constraints.

5.1 TARMS

A set of supporting assumptions, or *set of supports*, is a set of assumptions under which a value is valid. Since GMODS can derive many possible combinations of sets of supports for a value, a mechanism is needed to keep track of the sets of supports under which each value is valid. Additionally, data that are valid under particular sets of supports during one time interval may not be valid during another, and it is necessary to keep track of the time intervals during which values are valid under different sets of supports.

TARMS was developed to address these issues. It also assists the diagnostic problem solver to avoid making needless inferences that cannot contribute to a diagnosis, and to make the most useful inferences first. Overall, it performs four distinct functions:

- it generates the sets of supports for each value;
- it maintains a history of the time intervals in which each value is valid under different sets of supports;
- it assists the diagnostic problem solver by maintaining only the best sets of supports (the best sets of supports depend upon the algorithm in use, together with other considerations); and
- it assists the constraint propagator to avoid unnecessary inferences.

To perform these functions, TARMS deriving and updates *assumption histories* for each prediction about state variables. Assumption histories are an extension of Williams' value histories [20]. Value histories represent the behavior of a state variable over time as a contiguous, non-overlapping sequence of interval-value pairs which hold under one set of supporting assumptions. In contrast, an assumption history encodes a time-indexed chronology of different sets of supports under which a value is found to be valid for a particular state variable. TARMS maintains an assumption history for every variable-value combination derived during diagnostic reasoning.

The responsibility for temporal reasoning also resides within TARMS. A traditional constraint propagator generates values that are time independent, and TARMS uses temporal methods to determine the time intervals in which each value is valid. As noted earlier, the GMODS representation language facilitates this separation by differentiating between cause and effect relations and temporal behavior.

In the remainder of this section, we will consider how assumption histories are used to maintain a record of the time intervals in which values are valid under different sets of supports. In the following sections, we will then discuss the process used by TARMS to generate sets of supports under the two diagnostic algorithms, and we will then consider how the GMODS diagnostic problem solver uses TARMS to generate a diagnosis in an efficient manner.

5.2 ASSUMPTION HISTORIES

In the course of producing a diagnosis, a set of values will be inferred for each state variable within the GMODS circuit model. Given an assumption and values at antecedent state variables, a constraint is used to infer a value at a consequent state variable, when the constraint is one that is applicable in the assumed mode of operation.

Each inference is represented by a justification. A justification contains a consequent value, the sets of antecedents and the constraint that were used to infer the value, and the sets of supports under which the inference is valid. Each antecedent and consequent value is represented by a datum. A datum contains the value, the justifications that were used to infer the value, the justifications that use the value as an antecedent, and the sets of supports under which the value is valid. When the consequent datum is inferred, the set of antecedents is recorded within a justification, and the justification is attached to the datum. Since different constraints can be used to infer the datum at various points in the diagnostic reasoning process, any number of justifications may be attached to a datum.

As the behavior of a circuit changes over time, the sets of supports that explain each datum and justification may also change. An assumption history is used to keep track of which sets of supports explain each datum or justification over time. As illustrated in figure 5, several values at a state variable may be valid during a particular interval of time under different sets of supports, and the sets of supports that explain a particular value may change over time. The patterns in figure 5 and subsequent figures correspond to various sets of supports. The patterns vary, since the sets of supports may vary.

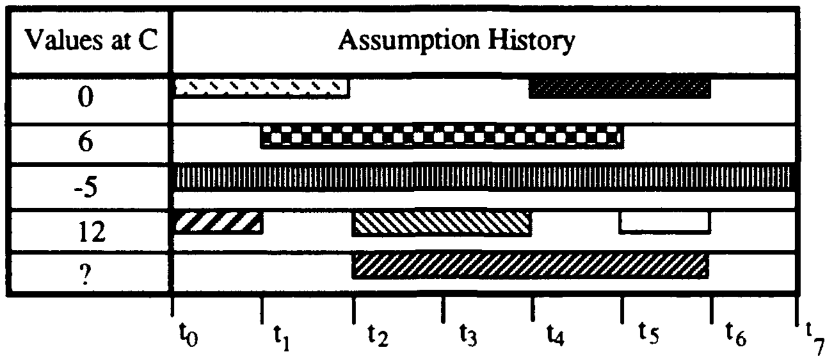

Figure 5. Assumption Histories for Values Inferred at a State Variable

GMODS uses assumption histories in two distinct ways, to maintain the sets of supports for each justification over time, and to maintain the best explanations for each datum. The best explanations for a datum are generated from the assumption histories of each justification for the datum. An assumption history for a justification is called a justification history and the assumption history for a datum is called a datum history.

Each element of an assumption history is a time-tagged explanation called an *occurrence*. Occurrences are used to capture the sets of supports that explain a justification during an interval of time, and to maintain the best sets of supports for a datum (from a particular pathway) during an interval of time. An occurrence has the following attributes:

- Origin - The datum or justification of the occurrence.
- Explanation - The sets of supports that explain the datum or justification.
- Time-interval - The time interval during which the sets of supports explain the datum.
- Weight - The number of fault mode assumptions contained in any of the sets of supports.

An occurrence that is associated with a datum also records the set of all occurrences for the same time interval which are associated with the justifications for the datum. This includes the ones that were not considered "best". Since any justification occurrence can become "best" at a later stage in the diagnostic reasoning process, all occurrences are cached here for efficient access.

Each value derived for an individual state variable is represented in GMODS as a *datum*. A datum has the following attributes:

- Value - A value inferred for a state variable.
- Justifications - A list of justifications that can be used to infer the value of the datum.
- Consequents - A list of justifications that include this datum as an antecedent.

- Assumption-history - A list of occurrences that support the datum at various intervals of time, indexed by pathway.

Finally, each justification has the following attributes:

- Consequent - The datum that is justified.
- Antecedents - The antecedents used to infer the consequent datum, including a list of antecedent datum and the assumption used to infer the consequent.
- Informant - The constraint used to infer the consequent.
- Assumption-history - A list of occurrences that support the justification at various intervals of time.

We will first consider how the assumption history for an individual justification is derived, and we will then describe the procedures for deriving the assumption history for a datum from its supporting justifications.

5.3 DERIVING THE ASSUMPTION HISTORY FOR A JUSTIFICATION

The justification history is derived using the following algorithm:

- Determine the temporal method implied by the constraint, and derive the temporal intervals.

- For each resulting time interval, derive the sets of supports for that interval based on antecedents at that interval, depending on the diagnostic algorithm in use. The antecedents that are used to determine the time intervals for an assumption history may be different than the antecedents used to derive the sets of supports.

- Make the assumption history concise by combining assumptions that are temporally adjacent and have the same sets of supports. This minimizes the number of assumptions that need to be maintained.

The temporal extent of occurrences in a justification history are determined by the temporal extents of occurrences in the datum histories of the justification's antecedents and the temporal method that is defined for the constraint. A temporal method describes the temporal behavior of a consequent assumption history by expressing the temporal relationship between the antecedents and the consequent of a justification, using Allen's temporal logic [1]. The central advantage of this modelling technique is that it allows temporal behaviors to be specified in terms of a small number of abstract rules. This methodology is particularly useful when diagnosing sequential circuits because the various components in the circuit could have very different temporal behaviors.

Temporal methods capture the similarities in temporal behavior that exist across digital devices, thus leading to concise circuit models and effective inference procedures.

Additionally, the use of temporal methods to describe temporal behavior is natural, given the separation within GMODS of value propagation and propagation of temporal information. The number of temporal methods that have been required to describe temporal behaviors within the GMODS models up to this point have been few; we have identified five distinct temporal methods to describe the temporal behavior of simple digital components: intersection, persistence, events, transitions, and delay.[3] We envision a small, fixed number of temporal methods that will be needed to describe the behavior of other digital circuits, and additional temporal methods may be added to the system as necessary.

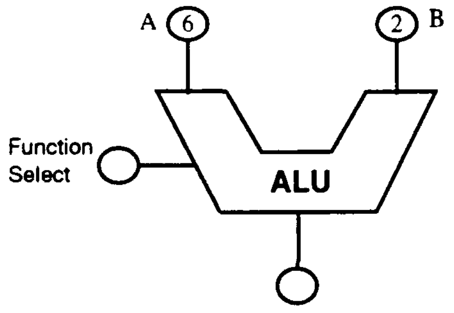

Figure 6. A Combinational Device

For components with combinational behaviors, such as the ALU in figure 6, the time intervals in which a justification is valid are computed by taking the intersection of the time intervals of all the antecedents. An occurrence is created for each time interval, and the best sets of supports for the occurrence are based on the best sets of supports for the corresponding occurrences within the assumption histories of each of the antecedents.

For example, assume that the ALU input A has a value of 2 during the intervals (t_0, t_4) and (t_5, t_6) and input B has a value of 6 during (t_1, t_3) and (t_5, t_7). We will also assume that the device has been selected for addition by C = 1 during (t_2, t_3) and (t_5, t_6), as shown in figure 7. The ***intersection method*** produces a new assumption history for the 8 that has been derived with temporal intervals which are valid in the intersection of the antecedents' intervals, namely (t_2, t_3) and (t_5, t_6).

[3] Our treatment of persistence has been influenced by Hamscher's work in the use of temporal reasoning within TINT [10].

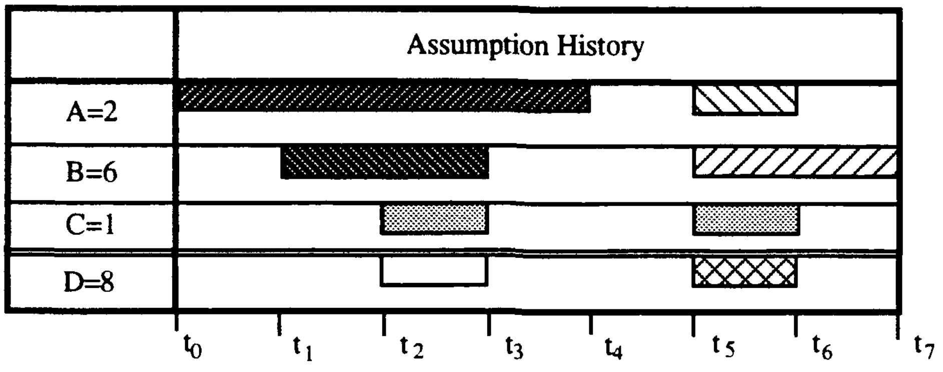

Figure 7. Creating an Assumption History using Intersection

Intersection cannot be used to derive the assumption history of a datum that persists within a memory device. Since a memory device, such as a latch or a register, has an internal state that persists, a datum may be valid within the internal memory of such a device when its antecedents are no longer valid. Three temporal methods can be used to describe the temporal behavior of a simple component with memory, such as the latch in figure 8. These methods allow GMODS to reason both forward and backward through time.

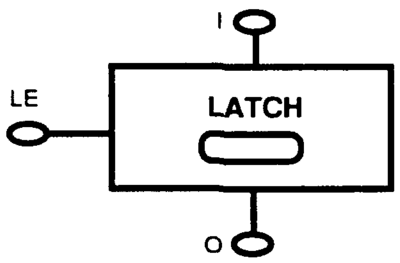

Figure 8. A Device with Memory

The first temporal method used to describe the behavior of a device with memory is the ***event method***. This method detects the event which causes a state transition. For example, it is used to determine when a transition from a load-enabled state to a non-load-enabled state occurs. This method recognizes the point in time when the control signal transitions, and is generally used to create an assumption history for a special transition variable. The transition variable is a local variable that maintains the current state of the device. Assumptions for the transition variable's assumption history are derived from the sets of supports from both the load enable and non-load enable value of the control signal.

The first and second temporal methods are closely related. The second method is a ***transition method***, which describes the behavior of a device immediately before or after an event. This method describes the relationship between the input and the internal memory of a memory device. The transition temporal method derives the temporal extent of a value in the internal memory of a device from its input during the first temporal interval after the device has transitioned from load-enabled to non-load-enabled. The extent of this interval is the smallest defined interval in the temporal logic. In a similar fashion, the temporal extent of the input of the device can be inferred from its internal memory one interval earlier than the transition.

The third method is the ***persistence method.*** Memory devices are typically controlled by control signals, such as the load-enable signal (LE) in figure 8. The control signal determines the interval during which the device's memory can be altered by inputs to the device. The persistence method computes the temporal extent of an occurrence for a value at the device's memory based on the extent of the control signal only. However, the sets of supports are derived from the sets of supports for the control signal and those for the previous memory value.

5.4 EXAMPLE

To illustrate the use of these temporal methods, consider a scenario where a 6 is observed at the input of the latch during the time interval (t_0, t_1), and the latch is load-enabled (LE = 1). The 6 is propagated to the latch-memory, and TARMS creates an occurrence on the history of the latch-memory that is valid during (t_0, t_1). During (t_1, t_4), LE = 0, causing the latch to no longer be load-enabled. TARMS uses the event method to create an occurrence on the history of the transition variable at time t_1. This occurrence indicates that the latch has transitioned from a load-enabled to a non-load-enabled state at t_1. Since a transition has occurred and GMODS is forward propagating, GMODS infers that the state of latch-memory immediately after the transition is the same as immediately before the transition. Using the transition method, an occurrence is created on the history of the latch-memory with a time interval of (t_1, t_2).

The assumption history of the latch-memory now intersects temporally with the history of (LE = 0), so the conditions for persistence are met. TARMS uses the persistence method to extend the occurrence on the history of the latch-memory from (t_1, t_2) to (t_1, t_4), which corresponds to the entire interval where the latch is not load-enabled. The latch continually outputs the contents of its memory, so the datum O = 6 at the output of the latch contains a history which is valid from (t_0, t_1) with the sets of supports under which the 6 was loaded, and from (t_1, t_4) with the sets of supports under which the value persists. This process is illustrated in figure 9.

The temporal methods work in a similar manner when reasoning backward in time. For example, suppose that a 6 is observed at the latch output during (t_2, t_3). Since the latch continually outputs the contents of its internal memory, a 6 can be inferred for latch-memory during (t_2, t_3).

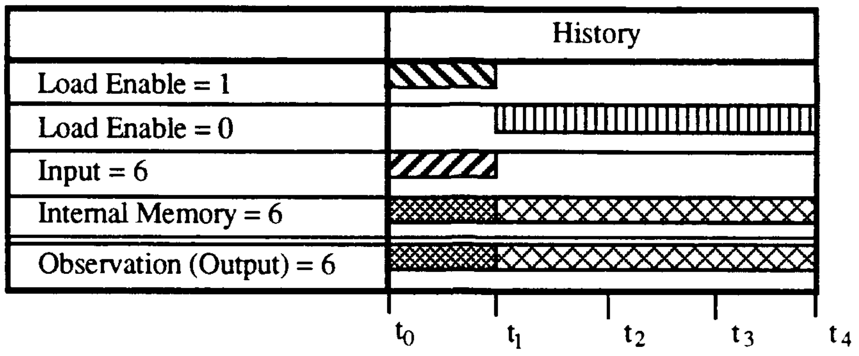

Figure 9. Reasoning Forward in a Device with Memory

As above, the latch is load-enabled during (t_0, t_1) and not load-enabled during (t_1, t_4). The conditions for persistence are met, so persistence method is used to extend the occurrence on the history of the latch-memory from (t_2, t_3) to (t_1, t_4), which corresponds to the interval when the latch is not load-enabled. Using the event method, a transition is detected at t_1, so an occurrence is created on the transition variable. This triggers the creation of the datum I = 6 with an occurrence for the time interval (t_0, t_1). This process is illustrated in figure 10.

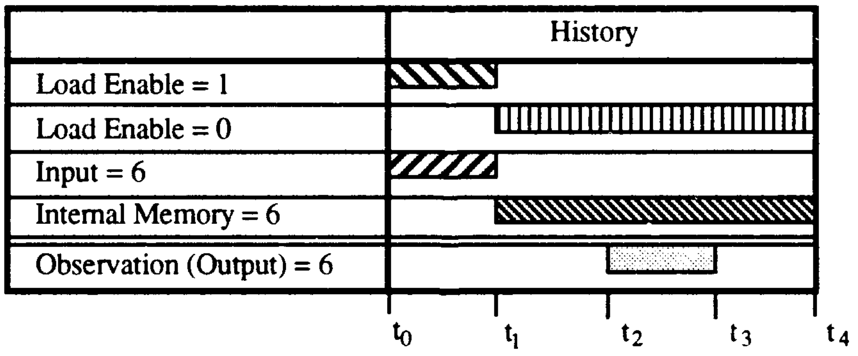

Figure 10. Reasoning Backward in a Device with Memory

5.5 DERIVING THE ASSUMPTION HISTORY FOR A DATUM

We will now describe how a datum history is derived from the justification histories that support the datum. A datum history is created or updated whenever the new justification history is derived, or the existing justification history has been updated. Separate datum histories are maintained for each pathway to the datum. (Although it is not necessary to maintain histories for each pathway when the conflict set algorithm is used, since GMODS is currently being used as a testbed to evaluate both algorithms, we will store information in this more general way.)

When a datum is supported by a single justification, the justification history is used for the datum history. However, when a datum receives a new justification, or the assumption history of one of its justifications is updated, then the history of the datum must be updated. The process of updating the datum history is discussed below.[4]

The first step is to create the maximal number of occurrences necessary for the updated datum history. Seven temporal relationships defined by Allen [1] are used in this process: starts, finishes, during, overlaps, meets, before, and equal. As an example, the overlaps relationship is shown in figure 11. Given two occurrences, one from the current datum history and one from the new justification history, these relations indicate that up to three new occurrences with certain time intervals are needed in the updated datum history to capture the evidence from both occurrences. To update the entire history, it is necessary to determine which of these relationships hold for successive pairs of the current datum history and the new justification history.

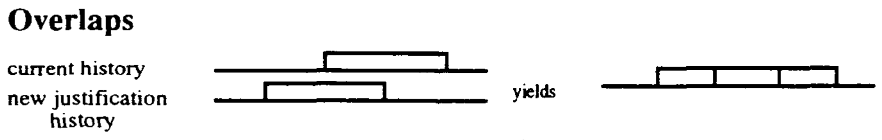

Figure 11. The Overlaps Temporal Relationship between Occurrences

The second step is to assign sets of supports to each occurrence derived for the new datum history. For each new occurrence, two occurrences, one from the current datum history and one from the new justification history, are selected based on the time interval of the new occurrence. If they both are rated "best", then the new occurrence includes the sets of supports from both occurrences. If not, then the best sets of supports from each occurrence are selected.

The third step merges occurrences that have the same sets of supports to form a concise history. If two adjacent occurrences have the same sets of supports, the occurrences can be

[4] This description of the process has been simplified for the sake of presentation. Obviously, more efficient algorithms are possible.

merged to form a single occurrence in the interval derived from the begin time of the first occurrence and the end time of the second occurrence. This step results in a more concise history.

Once a datum history is updated, its effects must be propagated to any consequent of that datum. These effects are propagated by visiting each of the consequent's justifications of that datum. A consequent's justification is a justification in which the updated datum is used as an antecedent. In this manner, these effects are recursively propagated throughout the dependency graph in TARMS, updating all relevant data assumption histories. This process is called *assumption propagation.* It can be streamlined by only updating the relevant time intervals of the histories, since the intervals that have not changed do not need to be updated. Modifications to a datum's explanation can therefore be propagated to the appropriate data without rederiving any inferences.

SECTION 6

CONSTRAINT PROPAGATION

The constraint propagator is responsible for propagating values through a GMODS model from inputs to outputs, and among outputs, to support the diagnostic reasoning process. It is made up of two subsystems: the *predicate evaluator* and the *consumer scheduler*. The consumer scheduler decides which inference will be made next by the constraint propagator. The constraint propagator then uses the predicate evaluator to make the inference.

6.1 THE PREDICATE EVALUATOR

The predicate evaluator is responsible for evaluating the individual predicates of a conditional constraint. A predicate can be evaluated in one of two modes: *generate* or *test*. If the predicate is ground (all state variables have values), then it is evaluated in test mode; the predicate evaluator determines whether the relationship holds among the values. Otherwise, the predicate is evaluated in generate mode; one of the state variables is left unspecified, and the predicate evaluator generates a value for it based on the given values. Predicates have various generators associated with them to do this. Generators are procedural information that can be used by the predicate evaluator to produce the value(s).

The predicate evaluator always return a boolean, *true* or *false*, and a binding (the value generated for the consequent state variable). In generate mode, *true* and some value is returned (or *false* and nil if a value could not be generated, e.g., divide by 0). In test mode, *true* or *false* and nil is returned. Depending upon the mode, both the boolean and the binding might be used.

For example, a constraint used to express the behavior of the ALU in figure 6, when it is performing addition, is:

```
(defconstraint alu-0 alu (alu-field alu-output input-a input-b)
      :applicable-mode-of-operation (working)
      :conditions ((*equal* (type-integer alu-field) 0))
      :consequents ((*sum-equal* (type-bit-vector alu-output)
                                  (type-bit-vector input-a)
                                  (type-bit-vector input-b)))
      :causal-flow alu-output
      :temporal-method intersection)
```

The predicates involved in this constraint are *equal* and *sum-equal*. The constraint is enforced only if the alu-field selects addition, when the predicate (*equal* (type-integer alu-field) 0) is true. The predicate evaluator is used to test whether the *equal* condition is true. If it is, then the constraint is used to make the *sum-equal* inference, to generate a new value for the appropriate state variable.

6.2 CONSUMERS

In the course of producing a diagnosis, there are typically a great number of possible inferences that can be made but that will not contribute to the diagnosis. To avoid making unnecessary inferences, we have developed a variation of the consumer architecture scheme described by de Kleer [4] to control the constraint propagator.

In a consumer architecture, all inferences that the constraint propagator can make are represented by *consumers*. A consumer represents a potential or pending inference. It is comprised of a set of antecedent data, a constraint, a consequent state variable, and an assumption history. Consumers are derived and then scheduled to be processed for the efficient performance of the diagnostic problem solver. When a consumer is processed, the constraint consequent is evaluated by the predicate evaluator to produce a datum for the consequent state variable, and the consumer is used to construct the justification for the new datum.

When a new datum is inferred for a state variable, many additional consumers can potentially be created. Each new consumer will consist of one of the constraints associated with the state variable, the new datum, and one of the datums already derived at each of the other antecedent nodes. When a consumer is created, an assumption history is created for the consumer using the assumption histories of the antecedents and the temporal method of the constraint. When assumption propagation takes place, it updates the assumption histories of consumers.

6.3 USING CONSUMERS TO AVOID USELESS INFERENCES

The consumer architecture guarantees that an inference will take place exactly once for each unique set of antecedents. In addition, since consumers are the only way to express potential inferences, we can sometimes avoid useless inferences by never creating certain consumers, and we can postpone processing other consumers until they will lead to a worthwhile inference. Some of the types of redundant or useless inferences avoided by using the consumer architecture are listed below:

- Inferring a consequent over and over again because the antecedents are valid at different time intervals. A consumer expresses the inference without respect to time. If an antecedent datum is found to be valid during a new time interval, the explanation for the consequent datum will be updated using assumption propagation, but no new consumer will be created and no additional constraint propagation will take place.

- Inferring a consequent if the same antecedent is derived from another reasoning pathway. If the datum that was previously derived is derived again from another pathway, no new consumers are created. All possible consumers for that datum were created when the datum was first derived. Of course, deriving the datum from the new pathway will invoke assumption propagation, which will propagate evidence to any consequents of the datum. This may cause the consumer scheduler to enable the previously generated consumers.

- Processing a conditional constraint whose condition is not valid. When consumers are being created, the condition of the constraint is tested by the predicate evaluator. If the condition is true, a consumer will be created for that set of antecedent data and the constraint. Otherwise, no consumer will be created. If a datum value is generated that would make the condition true, a consumer will be created at that time.

- Inferring a consequent that would not have any assumption history. A consumer is not processed if the temporal method specified for the constraint would yield no time intervals for the assumption history. For example, if a constraint used the intersection temporal method, then the time intervals in the assumption histories of all of the antecedents must intersect, otherwise the consequent would not be valid during any time interval.

- Handling contradictions. As constraint propagation takes place, contradictions may be discovered by the diagnostic problem solver. For instance, when using the minimal envisionment algorithm, a contradiction could occur if the sets of supports for the antecedents are inconsistent as a result of reconvergent fanout. Two divergent paths could converge downstream and be based on contradictory assumptions (e.g., A-working, A-faulted). When using the conflict set algorithm, contradictions detected by the diagnostic problem solver are used to create the conflict sets. Using either algorithm, further constraint propagation is avoided when the sets of supports for the consumer indicate a contradiction in all time intervals. An empty assumption history is used to indicate this situation.

Since a consumer's assumption history may change when the assumption history of an antecedent changes, a consumer with an empty assumption history may subsequently have one. Only consumers with non-empty assumption histories will be processed by the constraint propagator.

6.4 SCHEDULING CONSUMERS

Consumers are scheduled for processing based on the sets of supports within their assumption histories, which depends on the diagnostic algorithm in use. Since the conflict set algorithm only considers working mode behavior, the task of scheduling consumers is straightforward. All consumers except those which represent useless inferences must be processed, and they are processed in breadth-first order to reduce the amount of updating.

When using the minimal envisionment algorithm, the consumer scheduling process is more complex. Once consumers have been created, they are prioritized by a consumer scheduler in order of their weights. A consumer's weight is equal to the lowest weight of the occurrences in its assumption history. Consumers with a lower weight should be processed before those with a higher weight. This requirement tends to minimize the amount of updating that needs to be done to the assumption history of a datum. Consumers are indexed by their weight to facilitate this processing.

The weight of a consumer does not necessarily remain constant. As more constraint propagation takes place, this can cause TARMS to update an occurrence on the consumer's assumption history, which may cause a change in the weight of that consumer. As a consequence, a consumer's priority could change significantly (e.g., a consumer which was previously blocked or had a low priority could become high priority).

A contradiction consumer is assigned the *contradiction weight.* The contradiction weight is equal to one plus the number of components in the circuit, which is the highest possible weight. To block useless inferences, consumers that have this weight are never considered for processing.

By organizing the consumer schedule by weight, it is often possible to find the minimal diagnostic hypotheses by making fewer inferences. Since the consumer scheduler not only orders consumers according to weight, but also enforces a breath-first application of constraints, higher-weight consumers that will add nothing to the minimal diagnosis are not processed.

SECTION 7

DIAGNOSING FAULTS USING MINIMAL ENVISIONMENT

As described in section 3, the Geffner/Pearl algorithm traces through the dependency graph from an output value to identify the diagnostic hypotheses that are consistent with the observed behavior of the circuit. To implement this algorithm in GMODS, TARMS is configured to compute sets of supports which correspond precisely to these sets of diagnostic hypotheses. Rather than tracing through the dependency graph after constraint propagation has taken place, the sets of supports are cached *during* constraint propagation and updated as necessary. After forward and back propagation has been completed, the sets of supports computed for each output value represent the culmination of all of the evidence in the dependency graph. The sets of supports for the output values consistent with the observations are, therefore, the diagnostic hypotheses which explain the observed behavior of the circuit.

7.1 COMPUTING A DIAGNOSIS FOR A COMBINATIONAL CIRCUIT

A central function of TARMS is to combine the sets of supports associated with the justifications to generate the overall sets of supports under which the datum is valid.

The justifications which can provide support to a particular datum may include both the justifications for the datum itself, and the justifications for other *consistent* data at the state variable. In particular, the justifications which support ? also support any other derived value (e.g., the justifications for ? also support a 6 at the same state variable). In general, the consistency relationship describes which justifications should be shared. For instance, the value ? is consistent with an actual value (e.g., 6). Sets of supports for the actual value are based on the justification for the actual value, the justification for the ?, or a combination of the two. When the data at the state variable are of type tri-value vector, they form a partial order based on the consistency relation. For example, the tri-value vectors (? 1 1 0) and (0 ? 1 0) are both consistent with (0 1 1 0), but they are unrelated to each other. (? ? ? ?) is consistent with all three values.

To illustrate the process of generating sets of supports for a datum, we will examine the steps involved in generating the sets of supports for various nodes within the circuit of figure 12. Since sets of supports can be generated independently for different intervals of time, we will only consider the behavior of this circuit over one interval of time. (In this situation, since each history will contain only one occurrence, we will refer to the sets of supports in this occurrence as the sets of supports of datum or justification.)

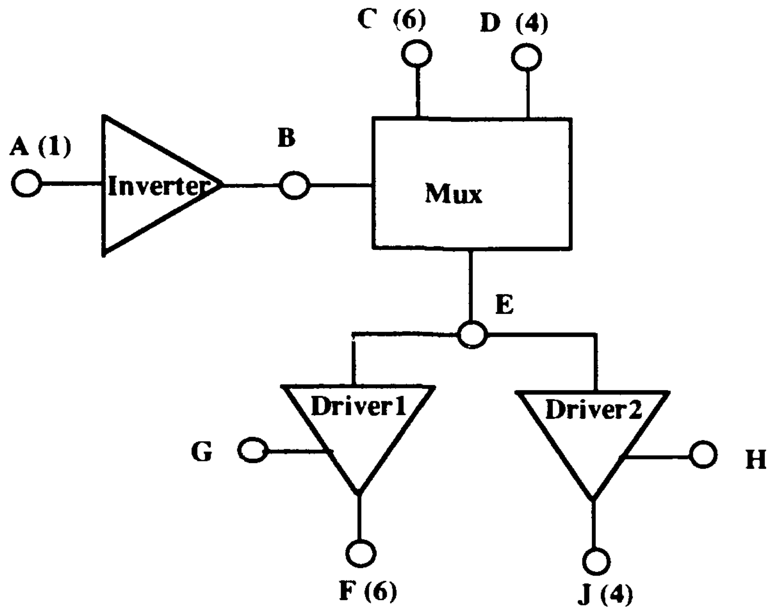

Figure 12. A Simple Multiplexer Circuit

In this simple multiplexer circuit, an inverter determines whether to transfer a signal from either input C or input D to the two drivers. When the inverter is working, it simply outputs the inverse of its input. It also has two fault modes: stuck-at-one and stuck-at-zero. The multiplexer and the drivers each have two modes of operation: working and the default fault mode. When the multiplexer is working, it transmits the value of the signal at the selected input port to the output port. A signal of 0 at the input-select port will cause the value at C to be transferred to the output port; otherwise, the value at D will be transferred. When the control port of either driver is 1 and the driver is working, it transfers the value at its input port to its output port. In this scenario, the inputs are A = 1, C = 6, and D =4 which should cause a 6 to be generated at both F and J. However, the observations at the outputs are F = 6 and J = 4.

The sets of supports for a datum are generated from the sets of supports associated with the justifications for the datum. For example, using forward propagation we can infer a 6 at node E with the justification:

[Justification: B = 0, C = 6, multiplexer-working]

and we can infer a 4 at the same node with the justification:

[Justification: B = 1, D = 4, multiplexer-working]

Each justification will have sets of supports associated with it. In the first case, the two assumptions that can support the antecedent datum B = 0 are inverter-stuck-at-zero and inverter-working, and so the sets of supports associated with the first justification are:

((inverter-working, multiplexer-working)
(inverter-stuck-at-zero, multiplexer-working)).

Similarly, the sets of supports associated with the second justification are:

((inverter-stuck-at-one, multiplexer-working)).

It is also possible to infer the datum E = ?. It has the justifications:

[Justification: B = 0, C = 6, multiplexer-faulted]
[Justification: B = 1, D = 4, multiplexer-faulted]

Since there is more than one justification the sets of supports for the datum E = ? are computed as the union of the sets of supports for each justification:

((inverter-working, multiplexer-faulted)
(inverter-stuck-at-one, multiplexer-faulted)
(inverter-stuck-at-zero, multiplexer-faulted))

As noted earlier, the datum E = ? is considered to be consistent with any other datum at E. Therefore, the sets of supports for E = ? are also valid for E = 6 and E = 4. Their sets of supports will include the sets of supports from E = ?:

E = 6:

((inverter-working, multiplexer-working)
(inverter-stuck-at-zero, multiplexer-working)
(inverter-working, multiplexer-faulted)
(inverter-stuck-at-one, multiplexer-faulted)
(inverter-stuck-at-zero, multiplexer-faulted))

E = 4:

((inverter-stuck-at-one, multiplexer-working)
(inverter-working, multiplexer-faulted)
(inverter-stuck-at-one, multiplexer-faulted)
(inverter-stuck-at-zero, multiplexer-faulted))

Forward propagation continues by generating a 6 at the outputs of the drivers. The justifications are [Justification: E = 6, G = 1, driver1-working] and [Justification: E = 6, H = 1, driver2-working] respectively. Lastly, we can generate a ? at outputs of the drivers. The justifications for F = ? are [Justification: E = 6, G = 1, driver1-faulted], [Justification: E = 4, G = 1, driver1-faulted], [Justification: E = ?, G = 1, driver1-working] and [Justification: E = ?, G = 1, driver1-faulted]. Similar justifications exist for J = ?.

Based on these justifications, the sets of supports for F = 6 are:

((inverter-working, multiplexer-working, driver1-working)
(inverter-stuck-at-zero, multiplexer-working, driver1-working)
(inverter-working, multiplexer-faulted, driver1-working)
(inverter-stuck-at-one, multiplexer-faulted, driver1-working)
(inverter-stuck-at-zero, multiplexer-faulted, driver1-working)
(inverter-working, multiplexer-working, driver1-faulted)
(inverter-stuck-at-one, multiplexer-working, driver1-faulted)
(inverter-stuck-at-zero, multiplexer-working, driver1-faulted)
(inverter-working, multiplexer-faulted, driver1-faulted)
(inverter-stuck-at-one, multiplexer-faulted, driver1-faulted)
(inverter-stuck-at-zero, multiplexer-faulted, driver1-faulted))

We will refer to the collection of the possible sets of supports for a datum as its *explanation*. An explanation is a disjunction of sets of support. For example, if a 6 was observed at node F in the example above, then one of the sets of supports listed in the explanation for the datum F = 6 must correspond to the actual state of the components upstream from F.

Generating the sets of supports for a datum can be computationally expensive. Fortunately, the cost of computing the sets of supports is reduced significantly by focusing on deriving the "best" sets of supports for a datum. These sets of supports are derived by choosing the best justification for the datum from each pathway.

Since the GMODS diagnostic algorithm assumes that each fault mode is equally likely, TARMS always chooses the justification that is based on the fewest number of fault mode assumptions. This justification is referred to as the minimal justification. In the example above, [Justification: B = 0, C= 6, multiplexer-working] is the minimal justification for E = 6 along the pathway from the multiplexer to node E, since it is based on the assumptions that both the multiplexer and inverter are working. The other justification from that pathway is not minimal since it is based on an assumption that the inverter is broken, and none of the justifications for E = ? are minimal. Similarly, [Justification: G = 1, E = 6, driver-working] is selected from the pathway from the driver. By selecting just the minimal justification from each pathway, the following explanation can be derived for F = 6:

((inverter-working, multiplexer-working, driver1-working)).

Thus, TARMS does not attempt to compute the complete explanation for a datum. Instead, it derives a subset of the complete explanation that includes only those sets of supports which assume that the fewest number of components have become faulted.

Of course, this above explanation does not make an assignment of a mode of operation to each component in the circuit and as such is not yet a diagnostic hypothesis. In order to complete the process we must perform back propagation. The value J = 4 must be back propagated through the circuit to the other output. Once back propagation is finished each value at F will have the sets of supports that contain an assignment of a mode of operation for every component in the circuit.

The first step is to generate E = 4 using [Justification: J = 4, H = 1, driver-working] which is valid under the sets of supports ((driver2-working)). The datum E = 4 is now supported by three justifications: two justifications from the pathway from the multiplexer to node E (via E = ?), and one justification from the pathway from the driver2 to node E. The datum E = ? is also now supported by an additional justification [Justification: J = 6, H = 1, driver2-faulted], but on a different pathway. The value E =4 is then forward propagated to F.

Additionally the sets of supports for F = 6 and F = ? must be updated by TARMS. The update is necessary because F = 6 depends upon E = 6 which now has evidence from the driver2 pathway via E = ?.

When there is more than one pathway TARMS derives the minimal sets of supports for each antecedent by taking the cross product of the minimal sets of supports of each pathway of the antecedent.

Note that the constraint propagator is not invoked in this updating process. The sets of supports from the same pathway as the direction of the inference must be excluded. Failure to exclude these sets of supports would cause circular reasoning. For example, the evidence on the pathway from driver2 is not used when updating the justifications used to derive values at J.

The minimal sets of supports for E =6 from the multiplexer pathway is ((inverter-working, multiplexer-working)) and from the driver2 pathway (using E = ?) is ((driver2-faulted)). Therefore, the sets of supports for [Justification: G = 1, E = 6, driver-working] are ((inverter-working, multiplexer-working, driver1-working, driver2-faulted)).

This justification is the minimal one available for F = 6. Since its sets of supports have changed, the sets of supports for F = 6 also change. F = 6 now has sets of supports which assign a mode of operation to every component is the circuit. These sets of supports therefore represent complete diagnostic hypotheses which can be returned by GMODS.

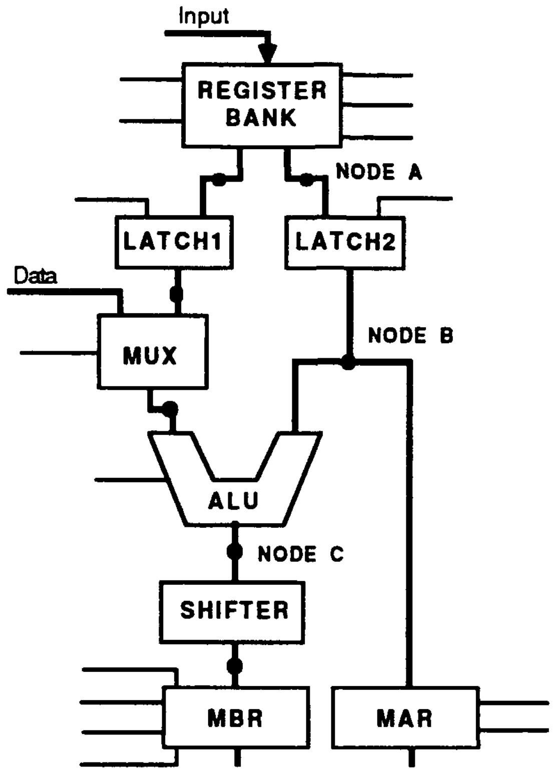

Figure 13. The Processor Test Case without Feedback

7.2 COMPUTING A DIAGNOSIS FOR A SEQUENTIAL CIRCUIT

Consider the following diagnostic scenario for the simple processor dataflow path shown in figure 13. A clock produces four pulses during each cycle. On the first clock pulse (time interval (t_0, t_1)), a 6 is loaded into the register bank and output to node A. During (t_0, t_4), a 2 is also input on the MUX data port and is selected as the MUX output. Latch2 is load-enabled during (t_1, t_2), and the contents of the latch memory are continually output to node B. Since the ALU function selected is addition, we expected to observe output values of MAR-output = 6 and MBR-output = 8 during (t_3, t_4). In this diagnostic scenario, however, the observed output values during (t_3, t_4) are MAR-output = 4 and MBR-output = 8. In addition to using this scenario to explain how GMODS can diagnose sequential circuits, will also show how the consumer scheduler is used by the constraint propagator.

To diagnose this circuit, the first step is to forward propagate values from circuit inputs to circuit outputs. To control this propagation, consumers are successively generated and processed under the control of the consumer scheduler. For example, when a 6 is inferred at node A, the following consumers are created:

C1. Antecedents: latch_in = 6, clock = 1, latch2-working
Constraint: LATCH-READ
Temporal Method: Intersection

C2. Antecedents: latch_in = 6, clock = 1, latch2-faulted
Constraint: LATCH-FAULTED-1
Temporal Method: Intersection

C3. Antecedents: latch_in = 6, clock = 0, latch2-faulted
Constraint: LATCH-FAULTED-1
Temporal Method: Intersection

Consequent State Variable	Constraint	Temporal Method	Value	Time Interval	Explanation
Latch-Input			6	(t_1, t_4)	((register-bank-working))
Latch-Input			?	(t_0, t_4)	((register-bank-faulted))
Latch-memory	Latch-read	Intersection	6	(t_1, t_2)	((register-bank-working, latch2-working))
Transition Variable	Latch-event	Event	T	(t_2, t_2)	Always-true
Latch-memory	Latch-transition	Transition	6	(t_2, t_3)	((register-bank-working, latch2-working))
	Latch-persistence	Persistence	6	(t_2, t_4)	((register-bank-working, latch2-working))
Latch-Output (Node B)	Latch-out	Intersection	6	(t_1, t_4)	((register-bank-working, latch2-working))
Latch-memory	Faulted-1	Normal	?	(t_0, t_4)	((register-bank-faulted, latch2-working) (register-bank-working, latch2-faulted))
Latch-Output (Node B)	Faulted-2	Normal	?	(t_0, t_4)	((register-bank-working, latch2-faulted))

Table 1. Inferences Made by Propagating a 6 through Latch2

Assumption histories are derived for each consumer, and the consumers are scheduled. When C1 is processed, a 6 is inferred for the latch memory. Since this is the first time a 6 is derived, the assumption history for the datum latch-memory = 6 is the same as the justification history. This 6 causes a consumer to be created which subsequently infers a 6 at node B on the pathway from Latch2 during (t_1, t_2) with the explanation ((register-bank-working, latch2-working)). Once a 6 is derived at the internal memory of Latch2, and the values for the control signals have been propagated, new consumers are created that process the event, transition, and persistence constraints as described in section 5.3.

When C2 and C3 are processed, a ? is derived with an assumption history which contains a minimal occurrence for the time interval (t_1, t_2) with the explanation ((register-bank-working, latch2-faulted)).

As shown in table 1, ? may also be inferred at the input to Latch2. The assumption history for this value would contain one minimal occurrence which is valid during (t_0, t_4) with the explanation ((register-bank-faulted)). Given this antecedent, the following consumers are created:

C4. Antecedents: latch_in = ?, latch2-working
Constraint: LATCH-FAULTED-1
Temporal Method: Intersection

C5. Antecedents: latch_in = ?, latch2-faulted
Constraint: LATCH-FAULTED-1
Temporal Method: Intersection

When these consumers are processed, two new justifications are added to support the ? at the internal memory of Latch2. C4 causes an assumption history to be derived with an occurrence with a weight of 1 during (t_0, t_4) with the explanation ((register-bank-faulted, latch2-working)). The assumption history for C5 contains a single occurrence with a weight of 2, which is valid during (t_0, t_4) with the explanation ((register-bank-faulted, latch2-faulted)).

Since multiple justifications support a value of ? at the internal memory of Latch2, the assumption history for ? must be updated with each new consumer processed. When C4 is processed, the assumption history is updated. The minimal occurrence for ? during (t_0, t_4) has an explanation of:

((register-bank-working, latch2-faulted)
(register-bank-faulted, latch2-working)).

Thus, both a 6 and ? are inferred at the latch-memory. These values are then propagated to node B. Afterward, the assumption history for a 6 at Node B on the pathway from Latch2 consists of two minimal occurrences, one which is valid during (t_0, t_1) under the explanation ((register-bank-working, latch2-faulted)), and a second which is valid during (t_1, t_4) under the explanation ((register-bank-working, latch2-faulted)).

The assumption history for ? on the pathway from Latch2 also consists of two minimal occurrences, one which is valid during (t_0, t_1) with the explanation ((register-bank-working, latch2-faulted)), and one which is valid during (t_1, t_4) with the explanation ((register-bank-working, latch2-faulted) (register-bank-faulted, latch2-working)).

Propagating forward through the MAR, two possible values can be inferred at the output of the MAR: 6 and ?. Since a 4 was observed at the output of the MAR, the only value consistent with the observation of a 4 during (t_3, t_4) is ?. The explanation for a ? during (t_3, t_4) is:

((register-bank-working, latch2-working, MAR-faulted)
(register-bank-faulted, latch2-working, MAR-working)
(register-bank-working, latch2-faulted, MAR-working))

Back propagation completes the diagnosis by combining evidence derived from all paths through the circuit. GMODS will back propagate from the MBR, through the Shifter and the ALU, to infer a 6 for node B (on the pathway from the ALU). This 6 has an assumption history that includes one occurrence which is valid during (t_3, t_4) with the explanation:

((ALU-working, Shifter-working, MBR-working, MUX-working)).

We then update the consequents of the 6 by updating the values inferred for the MAR-memory. One of these consequents is ? in the MAR-memory. The minimal occurrence for a ? in the MAR-memory has an antecedent of 6, and is based on the MAR being faulted. We update the assumption history at this ? by combining the evidence from the two pathways, and we obtain a ? with an assumption history containing one occurrence which is valid during (t_3, t_4) with the explanation:

((register-bank-working, latch2-working, MUX-working, ALU-working, shifter-working, MBR-working, MAR-faulted)).

This is the minimal occurrence for ? at the MAR-output. Any other occurrence that supports ? implies a double fault, and thus is not minimal.

We obtain the diagnosis by accessing the explanations for the predicted values which are consistent with the observed value during the observed time interval. In this case, the ? is the only predicted value consistent with the observed value of 4. Since the 4 was observed during (t_3, t_4), the diagnosis corresponds to the explanation for ? during (t_3, t_4), namely MAR-faulted.

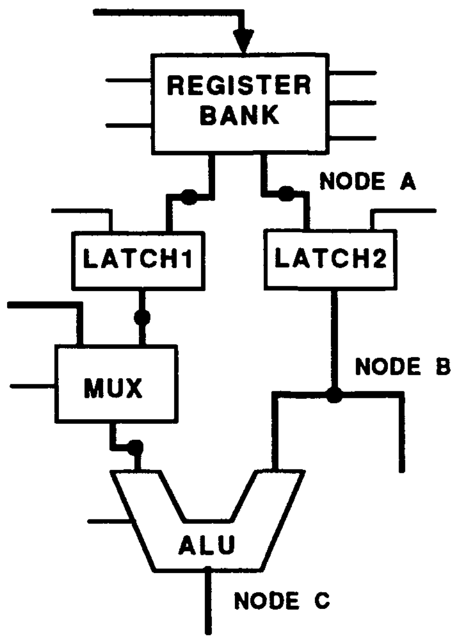

Figure 14. Highlighting Reconvergent Fanout in the Processor Circuit

7.3 HANDLING RECONVERGENT FANOUT

Reconvergent fanout occurs when divergent paths through a circuit reconverge at a component downstream. For example, in the part of the processor circuit shown in figure 14, the paths from the register bank reconverge at the ALU output (node C). When a circuit contains reconvergent fanout, the minimal sets of supports for any given antecedent may include assumptions that contradict the minimal sets of supports of another antecedent. If the basic minimal envisionment algorithm was used, it would lead to an incomplete set of diagnostic hypotheses. In the example, data at the inputs of the ALU could depend on contradictory assumptions register-bank-working and register-bank-faulted.

To illustrate this idea in more detail, consider the following scenario. The inputs to the circuit are the same as in section 7.2, but the output at the MAR is 4 and the output at the MBR is 7. The following justification generates a ? at node C:

J1. Antecedents: ALU-working, MUX-output = ?, Latch2-output =4
    Constraint: ALU-ADD
    Temporal Method: Intersection
    Evidence at time interval (t_3, t_4) for the antecedents:
        MUX-output = ?
          From MUX:
              Weight: 1
              Explanation: ((register-bank-working, latch1-working, MUX-faulted))
        Latch2-output =4
          From Latch2:
              Weight: 1
              Explanation: ((register-bank-working, latch2-faulted)
                            (register-bank-faulted, latch2-working))
          From MAR:
              Weight: 0
              Explanation: ((MAR-working))

The sets of supports for Latch2-output = 4 are based on evidence from two pathways – the pathway from Latch2 and the pathway from the MAR. The set of supports from the MAR is ((MAR-working)) due to the back propagation of the 4 observed at the output of the MAR. Since a 4 was not generated on the pathway from Latch2, the evidence from ? is used – which is ((register-bank-working, latch2-faulted), (register-bank-faulted, latch2-working)). The set of supports for the MAX-output = ? are ((MUX-faulted, latch1-working, register-bank-working)).

The sets of supports for this justification during (t_3, t_4) are ((register-bank-working, latch2-faulted, MAR-working, MUX-faulted, latch1-working), (register-bank-faulted, latch2-working, MAR-working, MUX-faulted, latch1-working, register-bank-working)). The second set contains a contradiction, so it is eliminated, yielding ((Reg-Bank-working, Latch2-faulted, MAR-working, MUX-faulted, Latch-1 working)). This set of supports has a weight of 2.

However, this is not the complete minimal set of supports for the justification during (t_3, t_4). There is another set, (Reg-Bank-faulted, Latch2-working, MAR-working, MUX-faulted, Latch-1 working), which also has a weight of 2. The problem arises because the evidence that would be used to produce this set of supports is based on non-minimal evidence for MUX-output = ?.

This suggests that the process of generating sets of supports for a datum at a node with reconvergent fanout should not be based solely on the minimal occurrences of each antecedent. The basic algorithm can be extended so that it backtracks to less minimal sets of supports of one or more of the antecedents. There may be other non-minimal sets of supports for antecedents that would lead to a resultant sets of support that are minimal and not contradictory.

For instance, ((MUX-faulted, Reg-Bank-faulted, Latch1-working), (MUX-faulted, Reg-Bank-working, Latch2-faulted)) is a less minimal set of supports of MUX-output = ?. By using these sets of supports, TARMS can generate the other set of supports that has a weight of 2.[5]

The extended algorithm requires TARMS to access all combinations of sets of supports for the antecedents which could yield the minimal weight for the consistent occurrence. Using the basic algorithm, the expected weight for the consequent's sets of supports is the sum of the weights of the antecedents' sets of supports. The weight of the resultant sets of supports can never be less than the maximum of the antecedents' occurrence weights, but if an antecedent's sets of supports contain the same fault assumptions, the weight of the resultant sets of supports will be less than the sum. To determine which sets of supports are actually minimal, all resultant sets of supports which *could* have that weight must be calculated. This is done using the following algorithm:

Step 1. Given the n assumption histories of the antecedents, determine the time intervals for the occurrences of the history for the consumer/justification.

Step 2. Iterate for each time interval generated in Step 1:

Step 2A. Iterate from i = 0 to the contradiction weight,

a. Find all occurrences of the antecedents whose combinations of weights could yield resultant sets of supports of weight i

b. Compute the sets of supports for each combination.

c. If all the resultant sets of supports are contradictions, then go to Step 2A and continue with next i

d. If there are some resultant sets of supports of weight i that are non-contradictory, then they are the minimal consistent sets of supports for this time interval, so go to 2 and do the next time interval

e. If there are sets of supports with higher weights, save the information, and go to Step 2A. These will be added later to the sets of supports found (if any) when the algorithm gets to the higher weight

[5] No sets of supports are generated which include MUX-working, because if the MUX is working in this scenario, it always reads from the input port and evidence from Latch1 is not included.

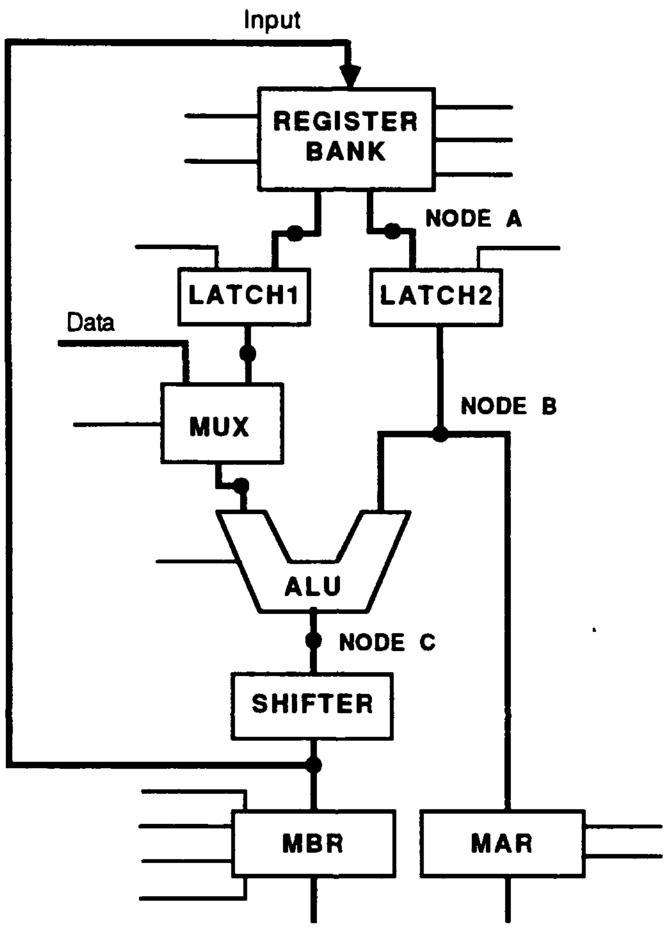

Figure 15. The Processor Test Case with Feedback

7.4 HANDLING FEEDBACK

This extension for handling reconvergent fanout is not sufficient for circuits with feedback, due to a cycling effect that occurs during constraint propagation. A consumer is normally processed when the antecedents have any support, either for the actual antecedent value or for the consistent values. Since ? is consistent with everything, actual values can always receive support from ?. In cases of feedback, the cycling or "ringing" that can result will enumerate all the possible values for ?. This cycling is prevented by limiting the constraint propagator to process consumers only when there is evidence for at least one actual antecedent value. Of course, when the antecedent value is ?, the consumer is processed if there is support for ?.

To illustrate the effects of this cycling problem, consider the circuit in figure 15. Again, we assume that each clock cycle consists of four clock pulses, and we will run the diagnostic scenario over two clock cycles. The scenario is similar to the one from section 7.2. A 6 is input into the MUX during (t_0, t_4) and a 2 is input during (t_4, t_8). From (t_0, t_4), the ALU performs the identity function, and from (t_4, t_8), addition is selected. The Register Bank is load enabled during (t_0, t_1) and output enabled to Node B during (t_5, t_8). Latch2 is load enabled during (t_1, t_2) and (t_5, t_6). The expected outputs during (t_7, t_8) are a 6 at the MAR output and an 8 at the MBR output.

When the circuit is working correctly, it is possible to generate an 8 at the input to the register bank during (t_4, t_8). The default failure behavior of the shifter can be used to generate a ? at the same node. The ? is valid during the time interval (t_0, t_8). Since ? is consistent with 8, the evidence for 8 is updated to include evidence from ?, making the 8 valid during (t_0, t_4). It is therefore possible to process the consumer [reg-bank-input = 8, clock = 1, register-bank-working]. As propagation continues, a 10 will be generated at the input to the register bank (by adding 2 to the 8). For the same reason as above, this 10 will be propagated through the register bank and eventually used to generate a 12 (by adding 2 to the 10). Nothing in the basic algorithm will stop this process from continuing as it tries to enumerate all the possible values for ?.

The enhanced algorithm avoids processing the consumer which loads the 8 into the register bank. The 8 is valid between (t_0, t_1) only through consistent evidence from the ?. If the algorithm ignores that evidence, then there is no intersection between the load-enable signal and the 8. As a result, the consumer will have an empty assumption history and will not be processed, and the unwanted cycling is prevented. At the same time, the evidence is still complete – any evidence that would be generated by propagating the 8 is generated when propagating the ? explicitly.

SECTION 8

DIAGNOSING FAULTS USING CONFLICT SETS

8.1 COMBINING SETS OF SUPPORTS

When using the conflict set algorithm in GMODS, the only consumers created are those that use the working assumption for the component; since no fault mode assumptions are propagated, every occurrence is of the same weight and consistent evidence from ? does not need to be taken into account. The TARMS method for determining the "best" sets of supports within each occurrence is also based on the use of parsimony rather than minimality.

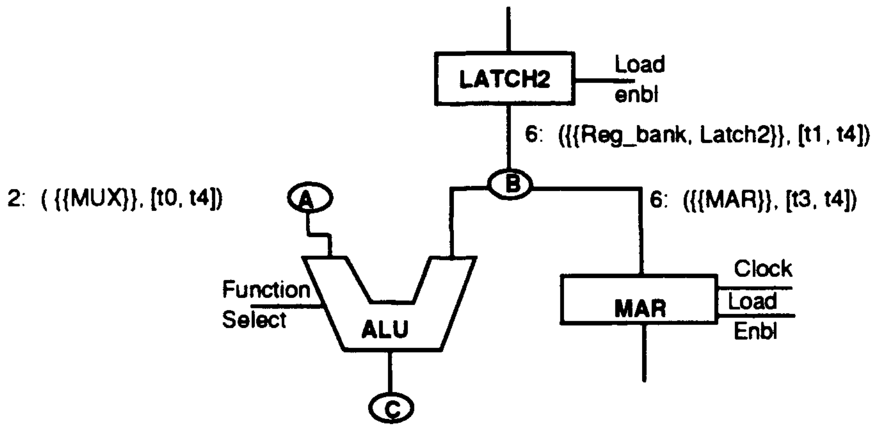

Figure 16. A Circuit Fragment

For example, assume that B = 6 and node A = 2 in figure 16. Using the addition constraint for the ALU, we can infer an 8 at node C, and we then need to determine the history for C = 8. As mentioned earlier, the first step in this process is to determine the history of all of the antecedents for the inference (A = 2) + (B = 6) => (C = 8). Since A = 2 has only one pathway, the history contributed by this antecedent is simply the history for that pathway, {((MUX-working)), (t_0, t_4)}. The datum B = 6 has two pathways, one from Latch2 and one from the MAR. The histories from each of these pathways must be combined into a single history for B = 6 that has occurrences for (t_1, t_3) and (t_3, t_4). The resulting sets of supports of the occurrence for (t_3, t_4) depends on the diagnostic algorithm used. When using minimal envisionment, the resulting sets of supports are created by combining the minimal sets of supports from each pathway using the cross product operation to produce ((Reg-bank-working, Latch2-working, MAR-working)). When using the conflict set algorithm, the sets of supports from each justification are kept separate, irrespective of pathway. The sets of supports are combined using the union operator, and then non-parsimonious sets are pruned, yielding ((Reg-bank-working, Latch2-working) (MAR-working)). Since there is only one justification for the time interval (t_1, t_3), both algorithms (concidentally) compute ((Reg-bank-working, Latch2-working)) for this time interval.

To compute the justification history for the inference (A = 2) + (B = 6) => (C = 8), the sets of supports contributed by each antecedent must then be combined for each relevant time interval. As in the minimal envisionment algorithm, the cross product operation is used. For the conflict set algorithm, however, non-parsimonious sets of supports must be pruned. For example, using the minimal envisionment algorithm, the result for (t_1, t_3) would be:

((Reg-bank-working, ALU-working, MUX-working, MAR-working, Latch2-working))

For the conflict set algorithm, the result would be:

((Reg-bank-working, ALU-working, MUX-working, Latch2-working)
(Reg-bank-working, ALU-working, MAR-working))

To compute the datum history for C = 8 using the conflict set algorithm, the appropriate sets of supports for each time interval are obtained from the justification histories and combined using the union operation. Non-parsimonious sets of supports are pruned.

8.2 COMPUTING THE DIAGNOSIS

When the occurrences for different values derived for a state variable overlap in time, a conflict has occurred. The sets of supports for the overlapping occurrences are combined using cross product operations to produce conflict sets. The most parsimonious conflict sets are maintained within a conflict set database. After constraint propagation has been completed and the diagnostic problem solver has found all of the conflicts, the diagnosis is computed from the conflict set database in one of two ways. If there is an intersection between the conflict sets, those components represent the possible single fault diagnoses. (There may be other parsimonious diagnoses consisting of multiple faults which will not be reported.) If no single faults can account for the discrepancy, or if a more complete diagnosis is desired, a parsimonious diagnosis can be computed by taking successive cross-products of the conflict sets.

In the following example, we will illustrate the use of the conflict set algorithm to diagnose a sequential circuit. Consider the following scenario for the circuit in figure 17. Values are clocked through the circuit over a four clock pulse cycle, and we will run the circuit for two cycles. A 6 is input into the MUX during (t_0, t_4) and a 2 is input to the MUX during (t_4, t_8). The ALU performs identity from (t_0, t_4) and performs addition from (t_4, t_8). The Register Bank is load enabled during (t_3, t_4) and output enabled to Node B during (t_5, t_8). Latch2 is load enabled during (t_1, t_2) and during (t_5, t_6). The expected outputs during (t_7, t_8) are a value of a 6 at the MAR output and a value of 8 at the MBR output. However, we observe that the circuit is faulted; although the expected result of 8 is observed at the MBR output, a discrepant value of 4 is observed at the MAR output.

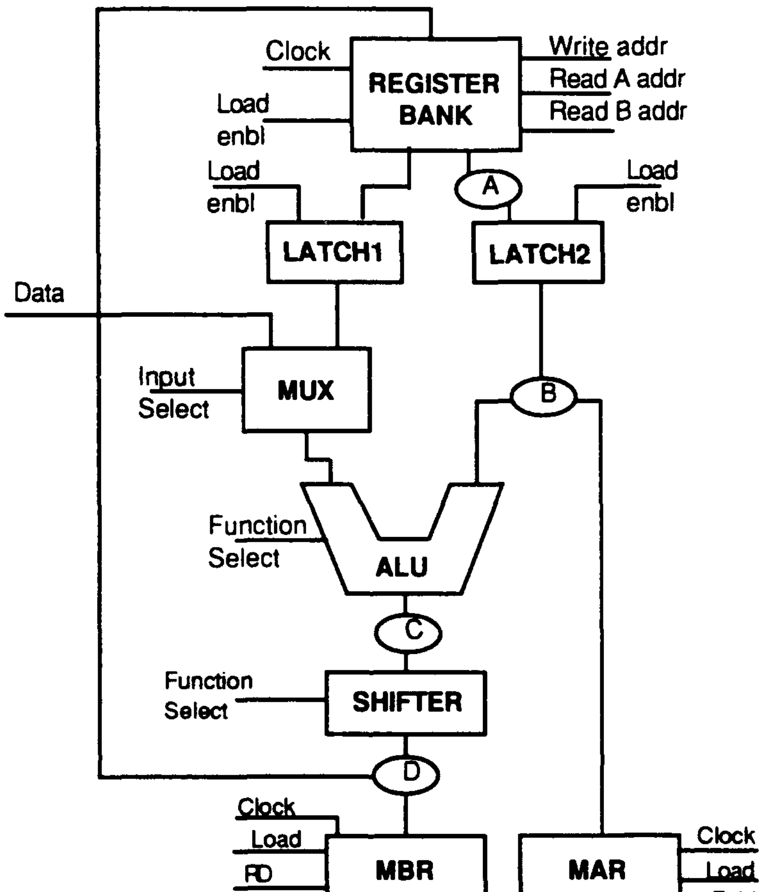

Figure 17. Processor Dataflow Path With Feedback

By forward propagation from the inputs, 6 is inferred for the MAR output. Since a 4 was observed, a contradiction is detected and the conflict set (Register_bank, Latch2, MAR) is generated. Since no other conflict sets are found during forward propagation, an initial diagnosis of a single fault can be obtained with the Register Bank, Latch2, and the MAR as diagnostic hypotheses.

The process of back propagating information among the outputs completes the diagnosis by combining evidence derived from all paths through the circuit. In this example, we can back propagate from the MBR, through the Shifter and the ALU, to infer a 6 at node B on the pathway from the ALU. The assumption history for this 6 includes one occurrence which is valid during (t_7, t_8) with the sets of supports:

(ALU, Shifter, MBR, MUX)

We then update the consequents of the 6 by updating the values obtained for MAR-memory using forward propagation. We update the assumption history of the 6 at node B by combining the evidence from the two pathways and then propagate this information to the MAR-output. This gives us another conflict of values at the output of the MAR. In addition to the original conflict set of (Register_bank, Latch2, MAR), we find a second conflict set (ALU, Shifter, MBR, MUX, MAR). An initial diagnosis of MAR is obtained by taking the intersection of the two conflict sets. A more complete diagnosis can be found by computing the cross product of the two conflict sets and then pruning all but the most parsimonious diagnoses to obtain:

((Register_bank, ALU) (Register_bank, Shifter) (Register_bank, MBR)
(Register_bank, MUX) (Latch2, ALU) (Latch2, Shifter) (Latch2, MBR)
(Latch2, MUX) (MAR))

SECTION 9

COMPARISION OF THE GMODS ALGORITHMS

By implementing the conflict set and minimal envisionment algorithms using the GMODS framework, we have demonstrated the capability of each algorithm to diagnose sequential circuits. In this section, we will evaluate their relative effectiveness in terms of processing performance, fault coverage, and plausibility.

9.1 PROCESSING PERFORMANCE

The processing required to implement each algorithm has the following steps in common:

- Forward propagate inputs through the circuit to obtain an initial diagnosis.
- Back propagate observed output values to refine the diagnosis.
- Maintain sets of supports to account for discrepancies between the observed and inferred values.

The computation required to produce a final diagnosis, however, depends upon the diagnostic algorithm employed. The minimal envisionment algorithm requires no additional processing; the sets of supports for the predicted output values consistent with the observations *are* the minimal diagnoses for the circuit. The conflict set algorithm does require additional computation – interesection operations are used to produce single fault diagnoses, and expensive cross product operations are necessary to produce more general parsimonious diagnoses. Situations can arise where a highly interconnected circuit topology with limited observability causes considerable ambiguity; in such cases the numbers and sizes of the conflict sets can become very large and the necessary cross product operations can become prohibitively expensive.

On the other hand, since the conflict set algorithm reasons about contradictions among the working behaviors of circuit components, it does not need to consider the effects of the default fault mode. This results in two major efficiencies when compared to the minimal envisionment algorithm. First, since ? is not generated, no consistency checking is required and much less updating takes place. Second, there is no need to retract commitments in circuits with reconvergent fanout. Since all occurrences are of the same weight and are equally valid, there are no non-minimal sets of supports and backtracking is unnecessary.

To provide a quantitative evaluation of the performance differences between the two diagnostic algorithms, we defined the following performance metrics:

- The observed processing time per interval. This metric is computed simply by dividing the recorded processing time by the number of clock intervals.

- The number of consumers processed per interval. Since processing consumers is the dominant and most expensive constraint propagation task, this metric measures the performance of the constraint propagator as a function of the number of clock intervals.

- The number of justifications updated per interval. The update-justification function includes (a) creating or updating the assumption history of the justification, and then (b) creating or updating the assumption history of the consequent value. Since this is the most expensive TARMS activity, this metric was chosen to measure the performance of TARMS.

In addition, since the Conflict Set algorithm can exhibit computational problems depending on the growth of the conflict set database, the number of conflict sets generated as a function of the number of clock intervals is also important for evaluating the Conflict Set algorithm.

GMODS performance statistics are shown in table 2. These statistics were generated by running GMODS on a Sun Sparcstation using the processor test case illustrated in sections 7 and 8. All of the statistics are averages from statistics recorded using four different diagnostic scenarios. The first column indicates the number of clock intervals simulated by the diagnostic problem

1. Conflict Set Algorithm:

CLOCK INTERVALS	PROCESSING TIME	CONSUMERS PROCESSED	JUSTIFICATIONS UPDATED	PROCESSING TIME /INTERVALS	CONSUMERS PROCESSED /INTERVALS	JUSTIFICATIONS UPDATED /INTERVALS
4	9.75	37	171	2.44	9.25	42.75
8	13.00	70	385	1.63	8.75	48.16
16	24.75	116	649	1.55	7.25	40.58
24	40.25	162	926	1.68	6.75	38.59
32	58.00	208	1206	1.81	6.50	37.68
40	75.75	254	1484	1.89	6.35	37.09

2. Minimal Envisionment Algorithm:

CLOCK INTERVALS	PROCESSING TIME	CONSUMERS PROCESSED	JUSTIFICATIONS UPDATED	PROCESSING TIME /INTERVALS	CONSUMERS PROCESSED /INTERVALS	JUSTIFICATIONS UPDATED /INTERVALS
4	37.25	86	433	9.31	21.56	108.13
8	141.75	204	2516	17.72	25.47	314.50

Table 2. GMODS Performance Statistics

solver. The second column indicates the processing time required to produce a diagnosis (in seconds). The third and fourth columns indicate the total number of consumers processed and the total number of justifications updated. The remaining columns correspond to the metrics above.

For the conflict set algorithm, these figures show that the numbers of consumers processed and justifications updated per interval decreased as the number of intervals increased. This is partly a consequence of using TARMS – because inferences are recorded by TARMS, fewer new inferences need to be made with each successive interval. Therefore, fewer consumers need to be processed and fewer justifications need to be updated. In addition, amount of processing required per interval is relatively constant. Initialization consumes a disproportionate amount of time, but this effect dissipates as the number of intervals increases. Subsequent increases in this metric are caused by additional processing required to access the assumption histories as they increase in size.

When running the conflict set algorithm, the number of conflict sets reached a stable value as the number of clock intervals increased. More generally, when a state variable is assigned conflicting values during one clock interval, it will tend to be assigned conflicting values during other intervals. The number of conflict sets therefore tends to be relatively independent of the number of clock intervals; it depends more directly upon the number of components and upon the topology of the circuit. When the size of the conflict set database does become large, single fault diagnoses can still be computed efficiently using intersection operations. On the other hand, the performance metrics for the minimal envisionment increased rapidly as a function of the number of clock intervals in all of the diagnostic scenarios we examined.

Since the performance metrics show that the performance of the conflict set algorithm is a stable function of the number of clock intervals, GMODS is ready for application to a more complex digital sequential circuit. Nonetheless, additional performance improvements are desirable to reason over larger number of time intervals. The processing time to produce a diagnosis could be reduced very substantially via the following techniques:

- Use different data structures for assumption histories. A linear search is currently required to extract evidence from an assumption history. A different data structure (e.g., an array) could provide more efficient access to the histories.

- Avoid unnecessary updating. Process inputs and outputs and propagate to reduce the number of updates that need to be made to assumption histories. Use symbolic representations of sequences (clock signals can be encoded as a repeating sequence of values, and maintaining a single assumption history for a sequence is much more efficient than maintaining histories for each value in the sequence). During back propagation, avoid generating evidence for time intervals that are not directly related to the observations.

- Remove generalities that permit GMODS to run both the Conflict Set and Minimal Envisionment algorithms.

9.2 FAULT COVERAGE

The two algorithms can also be compared on the basis of fault coverage. The conflict set algorithm generates the most parsimonious diagnoses for the circuit behavior. These explanations are actually a compact way to represent the full set of possible diagnoses, since all diagnoses subsumed by the parsimonious ones are also valid. (See [7] for a more detailed discussion of the implications of reporting parsimonious diagnoses.) When intersection operations are used, all of the parsimonious diagnoses may not be generated, and we cannot view the single fault diagnoses as a representation of the full set of valid diagnoses. However, single faults are considerably more common, and the algorithm for determining single fault diagnoses is considerably more efficient.

The minimal envisionment algorithm is based on the premises that all faults are equally likely and that the diagnoses containing a minimal *number* of fault assumptions are the most useful subset of the possible diagnoses. Unlike a parsimonious set of diagnoses, this set is not a compact representation of the full set of valid diagnoses. However, if new observations were used to rule out the currently minimal set, the necessary information is maintained within TARMS to efficiently calculate a new minimal set of diagnoses.

9.3 PLAUSIBILITY

The use of fault modes to refine the diagnosis has long been seen as desirable [6, 10, 18], and MITRE's first extension to the conflict set algorithm (GMODS-87) included fault modes [11]. However, this produced much larger sets of supports, more conflict sets, and longer processing times to produce parsimonious diagnosis. In the current GMODS conflict set algorithm, we have avoided using fault modes to reduce the number of conflicts to a more reasonable level.

The minimal envisionment algorithm must use fault modes to operate. In most situations, this involves only the default fault mode. The problem with the default fault modes is that they provide evidence for ?. Since such evidence also supports other values at the same state variable, the inclusion of fault modes creates a need for much more updating.

Other model-based diagnostic algorithms have used component fault probabilities to improve performance and improve plausibility. For example, Sherlock [6] demonstrates the feasibility of including fault modes to diagnose combinational circuits by using fault probabilities to explore likely diagnoses. It is able to compute the most likely diagnosis in five seconds for a combinational circuit with over 2000 components.

SECTION 10

CONCLUSIONS

The GMODS approach contributes to the state-of-the-art in model-based diagnostic reasoning in a number of ways: the GMODS model can be constructed directly from industry-standard VHDL design descriptions, GMODS is able to reason over multiple intervals of time within digital sequential circuits, the diagnostic problem solver is able to diagnose single and multiple faults with linear processing performance (in most cases) using the conflict set algorithm.

10.1 HARDWARE MODELING

The GMODS model used to support diagnostic reasoning can be constructed automatically from industry-standard VHDL design models. GMODS thus demonstrates the potential of model-based diagnostic reasoning to serve as an enabling technology to support concurrent engineering of the functional and diagnostic elements of the system design. When standard design models are used as a basis for diagnostic reasoning, they can be analyzed as an integral part of the design process to evaluate system performance in the presence of faults, and to evaluate the fault coverage provided by the diagnostics elements of the design. The design can then be improved incrementally as necessary, and after the system is fielded, the design model can be used to support field and depot-level troubleshooting activities.

Additional work is required to extend GMODS to accept standard design models that represent broad classes of realistic system designs. Presently, GMODS can use VHDL dataflow representations of behavior to diagnose faults at the register-transfer level within digital circuits. These register-transfer models were selected to develop the basic techniques for reasoning over time within sequential circuits. However, components such as registers and latches are among the most simple of the replaceable electronics components of modem systems, and before GMODS will be of utility to the engineering community, it will need to be able to represent and reason about more complex digital devices. To extend GMODS to diagnose faults within real circuit boards and systems, we must also generalize the VHDL subset supported by GMODS to accommodate the process-oriented behavioral models typically used to describe complex device behaviors in VHDL. It is possible that VHDL subsets defined to support design synthesis and performance modeling will form a suitable basis for a broader VHDL subset for diagnosis.

The internal GMODS representation language provides an additional contribution to hardware modeling – it can accommodate representations of analog and digital behaviors within a single declarative framework adequate to support diagnosis. In GMODS, the constraint and predicate constructs can be utilized to describe analog behavior where there is not always an identifiable direction of information flow. Indeed, predicates generally imply no causal direction; the intended causal direction is specified by a separate attribute within each constraint. To facilitate analog modeling, GMODS also permits multiple state variables to be associated with a node (e.g., the electrical currents flowing to a common node), and the GMODS constraint language permits node and loop equations to be attached directly to nodes within the model. Consequently, both analog and digital component models could potentially be translated into GMODS to support diagnosis of mixed analog/digital systems.

10.2 REASON MAINTENANCE

TARMS serves as an enabling technology to permit diagnostic reasoning over multiple intervals of time within sequential circuits. Unlike previous reason maintenance approaches which incorporate temporal information by maintaining value histories, TARMS maintains assumption histories for each value. A major benefit of using assumption histories is that existing data need not be rederived whenever a new assumption is made. For example, 4 and 5 may serve as antecedents to an addition constraint that is used to infer a 9, based on a combination of the sets of supports obtained from the constraint and the antecedents. Since 4 plus 5 will always equal 9, this inference does not need to be rederived when the sets of supports for the antecedents change – the assumption history for the datum 9 *will maintain* the different sets of supports under which the 9 is valid at various times. As additional information is propagated through the network over time, the consumer architecture prevents existing inferences from being rederived, and TARMS updates the relevant intervals in the assumption histories to reflect the new information.

TARMS also supports reasoning about the behavior of components with memory. When a value persists, it does not change, but the time intervals in which it is valid and its sets of support can change significantly. The separation of value propagation and temporal behavior naturally supports persistence. A value that persists does not need to be rederived for each time interval; using the temporal reasoning mechanisms of TARMS, the different sets of supports for each time interval are computed directly.

10.3 DIAGNOSTIC REASONING

GMODS has made significant progress toward satisfying the goals outlined within the introduction. GMODS can provide thorough fault coverage since it is able to diagnose unanticipated faults and multiple faults. When running the conflict set algorithm, GMODS is able to meet the efficiency goal in most cases, and the conflict set algorithm appears to be well behaved as a function of the number of time intervals. When using the minimal envisionment algorithm, explicit fault modes can be incorporated into the model, and GMODS has the potential to avoid generating implausible diagnostic hypothesis. Unfortunately, even after exploiting the consumer architecture to avoid unnecessary inferences, the minimal envisionment algorithm has shown unacceptable processing performance when reasoning over time within sequential circuits.

Since GMODS is able to reason about behavior over time within sequential circuits by using the conflict set algorithm, it has made a significant advance toward "scaling up" to address realistic applications. More work is clearly required to adapt GMODS to diagnose faults within realistic electronics boards and systems. The appropriate next step is to identify a series of realistic digital circuit boards of increasing complexity and to incrementally refine the GMODS approach as it is applied to each successive board.

LIST OF REFERENCES

1. Allen, J. F., "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM,* Vol. 26, November 1983.

2. Davis, R., "Diagnostic Reasoning Based on Structure and Behavior," *Artificial Intelligence,* Vol. 24, 1984.

3. de Kleer, J., "An Assumption-based TMS," *Artificial Intelligence,* Vol. 28, 1986.

4. de Kleer, J., "Problem Solving with the ATMS," *Artificial Intelligence,* Vol. 28, 1986.

5. de Kleer, J. and B. Williams, "Diagnosing Multiple Failures," *Artificial Intelligence,* Vol. 32, 1987.

6. de Kleer, J. and B. Williams, "Diagnosis with Behavioral Modes," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, August 1989.

7. de Kleer, J., A. Mackworth, and R. Reiter, "Characterizing Diagnoses," *Proceedings of the Eighth National Conference on Artificial Intelligence*, August 1990.

8. Geffner, H. and J. Pearl, "An Improved Constraint Propagation Algorithm for Diagnosis," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, August 1987.

9. Genesereth, M., "The Use of Design Descriptions in Automated Diagnosis," *Artificial Intelligence,* Vol. 24, 1984.

10. Hamscher, W. C., "Model-Based Troubleshooting of Digital Systems," *MIT Artificial Intelligence Laboratory Technical Report AI-TR-1074*, August 1988.

11. Holtzblatt, L. J., "Diagnosing Multiple Failures Using Knowledge of Component State," *Fourth IEEE Conference on Artificial Intelligence Applications*, March 1988.

12. *IEEE Standard VHDL Language Reference Manual,* IEEE Std. 1076-1987, March 1988.

13. Marcotte, R. A., L. J. Holtzblatt, R. C. Labonté, and R. L. Piazza, "A Model-based Approach for Diagnosing Launch System Hardware," *Ninth International Workshop on Expert Systems and their Applications*, June 1989.

14. Marcotte, R. A., M. J. Neiberg, and J. M. Schoen, "System-level Applications and Research Directions for Model-based Diagnostic Reasoning," *Second AAAI Workshop on Model-based Reasoning*, July 1990.

15. Marcotte, R. A., M. J. Neiberg, R. L. Piazza, and L. J. Holtzblatt, "Model-based Diagnostic Reasoning using VHDL," *Performance and Fault Modeling with VHDL (edited by J. M. Schoen)*, Prentice Hall, October 1991.

16. Pan, J., "Qualitative Reasoning with Deep-level Mechanism Models for Diagnoses of Mechanism Failures," *First IEEE Conference on Artificial Intelligence Applications*, 1984.

17. Scarl, E. A., J. R. Jamieson, and C. I. Delaune, "Diagnosis and Sensor Validation Through Knowledge of Structure and Function," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 17, No. 3, 1987.

18. Struss, P. and O. Dressler, "Physical Negation: Integrating Fault Modes into the General Diagnostic Engine," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, August 1989.

19. Sussman, G. and G. Steele, "Constraints - A Language for Expressing Almost-Hierarchical Descriptions," *Artificial Intelligence*, 1980.

20. Williams, B., "Doing Time: Putting Qualitative Reasoning on Firmer Ground," *Proceedings of the Sixth National Conference on Artificial Intelligence*, August 1986.

DISTRIBUTION LIST

INTERNAL

A010

R. D. Haggarty
C. J. Whiting

D010

H. S. Sorenson

D011

R. W. Jacobus

D052

D. R. Mangsen
T. F. Roome
T. Szczerbinski

D058

P. E. Barck
R. J. Byrne
J. R. Spurrier
C. W. Traber

D074

T. K. Backman
L. J. Holtzblatt
M. H. Starsiak

D080

J. M. Schoen

D090

A. Chu
H. M. Cronson

G010

V. A. DeMarines

G110

H. A. Bayard
E. H. Bensley
E. L. Lafferty
S. D. Litvintchouk
P. S. Tasker

G111

M. P. Chase
B. A. Goodman
E. J. Green
S. E. Hansen
R. A. Marcotte (20)
M. T. Maybury
M. J. Neiberg (5)
R. L. Piazza (5)
A. S. Rosenthal
M. S. Sayko
A. M. Tallant
M. B. Vilain
K. C. Warren
J. P. L. Woodward

H124

L. P. Seidman
L. Williams

PROJECT

Rome Laboratory
Griffiss Air Force Base, NY 13441

Lt. T. Pettinato (15)
RL/RBRP

Warner Robins Air Logistics Center
Robins Air Force Base, GA 31098

R. Colson
WR-ALC/LYPRA

R. Sveyda
WR-ALC/LYPAC

Wright Laboratory
Wright Patterson Air Force Base, OH 45433

W. Russell
WL/MTIB

EXTERNAL

Rockwell International Corporation
4311 Jamboree Road
Newport Beach, CA 92658-8902

D. J. Moder
M.S. 501-365

National Aeronautics and Space Administration
Kennedy Space Center, FL 32899

T. Davis
PT-TPO